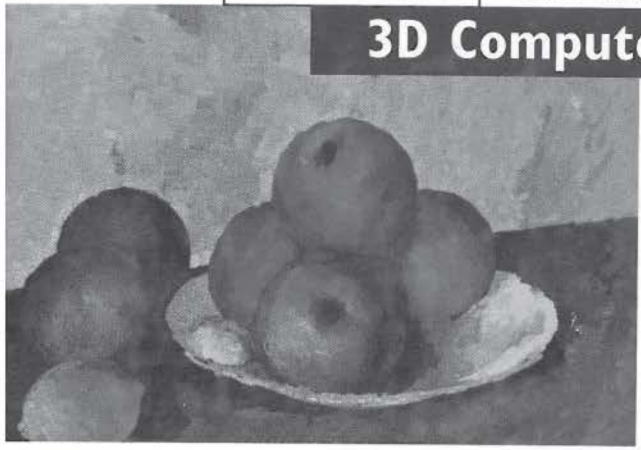
COMPUTER GRAPHICS

Alan Watt Third Edition

3D Computer Graphics

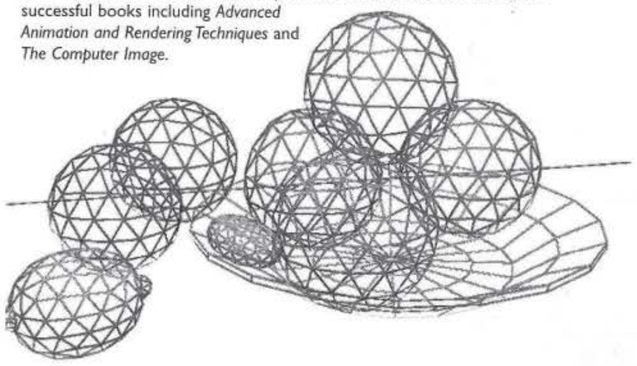


This book will enable you to master the fundamentals of 3D computer graphics. As well as incorporating recent advances across all of computer graphics it contains new chapters on

- Advanced radiosity
- Animation
- Pre-calculation techniques

and includes a CD containing a 400 image study.

Alan Watt, based at the University of Sheffield, is the author of several successful books including *Advanced Animation and Rendering Techniques* and *The Computer Image*.



This is a third edition of a book that deals with the processes involved in converting a mathematical or geometric description of an object (a computer graphics model) into a visualisation (a 2D projection) that simulates the appearance of a real object. Algorithms in computer graphics mostly function in a 3D domain and the creations in this space are then mapped into a 2D display or image plane at a late stage in the overall process. Traditionally computer graphics has created pictures by starting with a very detailed geometric description, subjecting this to a series of transformations that orient a viewer and objects in 3D space, then imitating reality by making the objects look solid and real – a process known as rendering. Nowadays this is proving insufficient for the new demands of moving computer imagery and virtual reality and much research is being carried out into how to model complex objects, where the nature and shape of the object changes dynamically and into capturing the richness of the world without having to model every detail explicitly. Such efforts are resulting in diverse synthesis methods and modelling methods.

This book will act as a complete resource for anyone interested in 3D modelling, providing detailed coverage of both realistic and non-realistic images.

ISBN 0-201-39855-9
90000>
9 780201 398557

PEARSON
Addison Wesley

Back cover: non-photo realistic rendering by Dan Teece; front cover: photo realistic rendering by Lightwork Design

www.pearson-books.com

Alan Watt 3D Computer Graphics Third Edition

PEARSON
Addison Wesley

Alan Watt Third Edition

3D Computer Graphics

PEARSON
Addison Wesley

Pearson Education Limited
Edinburgh Gate
Harlow
Essex CM20 2JE
England

and Associated Companies throughout the world

Visit us on the World Wide Web at:
www.pearsoneduc.com

First published 1989
Second edition 1993
This edition first published 2000

ISBN 0 201 39855 9

British Library Cataloguing-in-Publication Data
A catalogue record for this book can be obtained from the British Library.

Library of Congress Cataloguing-in-Publication Data
Available from the publisher.

Typeset by 42
Printed and bound in The United States of America

10 9 8 7
07 06 05 04 03

Para Dionéa
a garota de Copacabana

Contents

Preface

This is the third edition of a book that deals with the processes involved in converting a mathematical or geometric description of an object – a computer graphics model – into a visualization – a two-dimensional projection – that simulates the appearance of a real object. The analogy of a synthetic camera is often used and this is a good allusion provided we bear in mind certain important limitations that are not usually available in a computer graphics camera (depth of field and motion blur are two examples) and certain computer graphics facilities that do not appear in a camera (near and far clipping planes).

Algorithms in computer graphics mostly function in a three-dimensional domain and the creations in this space are then mapped into a two-dimensional display or image plane at a late stage in the overall process. Traditionally computer graphics has created pictures by starting with a very detailed geometric description, subjecting this to a series of transformations that orient a viewer and objects in three-dimensional space, then imitating reality by making the objects look solid and real – a process known as rendering. In the early 1980s there was a coming together of research – carried out in the 1970s into reflection models, hidden surface removal and the like – that resulted in the emergence of a *de facto* approach to image synthesis of solid objects. But now this is proving insufficient for the new demands of moving computer imagery and virtual reality and much research is being carried out into how to model complex objects, where the nature and shape of the object changes dynamically and into capturing the richness of the world without having to explicitly model every detail. Such efforts are resulting in diverse synthesis methods and modelling methods but at the moment there has been no emergence of new image generation techniques that rival the pseudo-standard way of modelling and rendering solid objects – a method that has been established since the mid-1970s.

So where did it all begin? Most of the development in computer graphics as we know it today was motivated by hardware evolution and the availability of new devices. Software rapidly developed to use the image producing hardware. In this respect the most important development is the so-called raster display, a device that proliferated in the mass market shortly after the development of the PC. In this device the complete image is stored in a memory variously called a

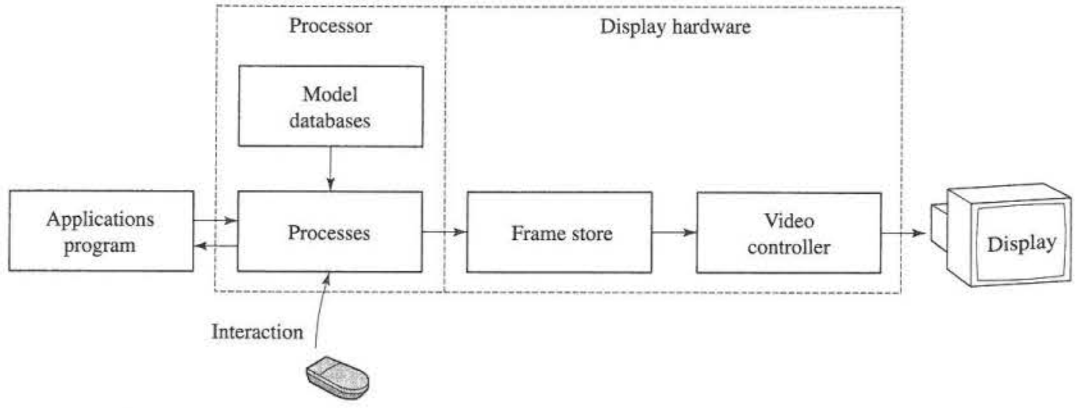


Figure P.1
The main elements of a graphics system.

frame store, a screen buffer or a refresh memory. This information – the discretized computer image – is continually converted by a video controller into a set of horizontal scan lines (a raster) which is then fed to a TV-type monitor. The image is generated by an application program which usually accesses a model or geometric description of an object or objects. The main elements in such a system are shown in Figure P.1. The display hardware to the right of the dotted line can be separate to the processor, but nowadays is usually integrated as in the case of an enhanced PC or a graphics workstation. The raster graphics device overshadows all other hardware developments in the sense that it made possible the display of shaded three-dimensional objects – the single most important theoretical development. The interaction of three-dimensional objects with a light source could be calculated and the effect projected into two-dimensional space and displayed by the device. Such shaded imagery is the foundation of modern computer graphics.

The two early landmark achievements that made shaded imagery possible are the algorithms developed by Gouraud in 1971 and Phong in 1975 enabling easy and fast calculation of the intensities of pixels when shading an object. The Phong technique is still in mainstream use and is undoubtedly responsible for most of the shaded images in computer graphics.

A brief history of shaded imagery

When we look at computer graphics from the viewpoint of its practitioners, we see that since the mid-1970s the developmental motivation has been photo-realism or the pursuit of techniques that make a graphics image of an object or scene indistinguishable from a TV image or photograph. A more recent strand of the application of these techniques is to display information in, for example, medicine, science and engineering.

The foundation of photo-realism is the calculation of light–object interaction and this splits neatly into two fields – the development of local reflection models and the development of global models. Local or direct reflection models only consider the interaction of an object with a light source as if the object and light were floating in dark space. That is, only the first reflection of light from the object is considered. Global reflection models consider how light reflects from one object and travels onto another. In other words the light impinging on a point on the surface can come either from a light source (direct light) or indirect light that has first hit another object. Global interaction is for the most part an unsolved problem, although two partial solutions, ray tracing and radiosity, are now widely implemented.

Computer graphics research has gone the way of much modern scientific research – early major advances are created and consolidated into a practical technology. Later significant advances seem to be more difficult to achieve. We can say that most images are produced using the Phong local reflection model (first reported in 1975), fewer using ray tracing (first popularized in 1980) and fewer still using radiosity (first reported in 1984). Although there is still much research being carried out in light–scene interaction methodologies much of the current research in computer graphics is concerned more with applications, for example, with such general applications as animation, visualization and virtual reality. In the most important computer graphics publication (the annual SIGGRAPH conference proceedings) there was in 1985 a total of 22 papers concerned with the production techniques of images (rendering, modelling and hardware) compared with 13 on what could loosely be called applications. A decade later in 1995 there were 37 papers on applications and 19 on image production techniques.

Modelling surface reflection with local interaction

Two early advances which went hand-in-hand were the development of hidden surface removal algorithms and shaded imagery – simulating the interaction of an object with a light source. Most of the hidden surface removal research was carried out in the 1970s and nowadays, for general-purpose use, the most common algorithm is the Z-buffer – an approach that is very easy to implement and combine with shading or rendering algorithms.

In shaded imagery the major prop is the Phong reflection model. This is an elegant but completely empirical model that usually ends up with an object reflecting more light than it receives. Its parameters are based on the grossest aspects of reflection of light from a surface. Despite this, it is the most widely used model in computer graphics – responsible for the vast majority of created images. Why is this so? Probably because users find it adequate and it is easy to implement.

Theoretically based reflection models attempt to model reflection more accurately and their parameters have physical meaning – that is they can be measured for a real surface. For example, light reflects differently from an isotropic surface, such as plastic, compared to its behaviour with a non-isotropic surface

such as brushed aluminium and such an effect can be imitated by explicitly modelling the surface characteristics. Such models attempt to imitate the behaviour of light at a 'milliscale' level (where the roughness or surface geometry is still much greater than the wavelength of light). Their purpose is to imitate the material signature – why different materials in reality look different. Alternatively, parameters of a model can be measured on a real surface and used in a simulation. The work into more elaborate or theoretical local reflection models does not seem to have gained any widespread acceptance as far as its implementation in rendering systems is concerned. This may be due to the fact that users do not perceive that the extra processing costs are worth the somewhat marginal improvement in the appearance of the shaded object.

All these models, while attending to the accurate modelling of light from a surface, are local models which means that they only consider the interaction of light with the object as if the object was floating in free space. No object–object interaction is considered and one of the main problems that immediately arises is that shadows – a phenomenon due to global interaction – are not incorporated into the model and have to be calculated by a separate 'add-on' algorithm.

The development of the Phong reflection model spawned research into add-on shadow algorithms and texture mapping, both of which enhanced the appearance of the shaded object and tempered the otherwise 'floating in free space' plastic look of the basic Phong model.

Modelling global interaction

The 1980s saw the development of two significant global models – light reflection models that attempt to evaluate the interaction between objects. Global interaction gives rise to such phenomena as the determination of the intensity of light within a shadow area, the reflection of objects in each other (specular interaction) and a subtle effect known as colour bleeding where the colour from a diffuse surface is transported to another nearby surface (diffuse interaction). The light intensity within a shadow area can only be determined from global interaction. An area in shadow, by definition, cannot receive light directly from a light source but only indirectly from light reflecting from another object. When you see shiny objects in a scene you expect to see in them reflections of other objects. A very shiny surface, such as chromium plate, behaves almost as a mirror taking all its surface detail from its surroundings and distorting this geometrically according to surface curvature.

The successful global models are ray tracing and radiosity. However, in their basic implementation both models only cater for one aspect of global illumination. Ray tracing attends to perfect specular reflection – very shiny objects reflecting in each other, and radiosity models diffuse interaction which is light reflecting off matte surfaces to illuminate other surfaces. Diffuse interaction is common in man-made interiors which tend to have carpets on the floor and matte finishes on the walls. Areas in a room that cannot see the light source are illuminated by diffuse interaction. Mutually exclusive in the phenomena they model, images created by both methods tend to have identifying 'signatures'. Ray-traced images are notable for perfect recursive reflections and super sharp refraction. Radiosity images are usually of softly-lit interiors and do not contain specular or shiny objects.

Computer graphics is not an exact science. Much research in light–surface interaction in computer graphics proceeds by taking existing physical models and simulating then with a computer graphics algorithm. This may involve much simplification in the original mathematical model so that it can be implemented as a computer graphics algorithm. Ray tracing and radiosity are classic examples of this tendency. Simplifications, which may appear gross to a mathematician, are made by computer graphicists for practical reasons. The reason this process 'works' is that when we look at a synthesized scene we do not generally perceive the simplifications in the mathematics unless they result in visible degeneracies known as aliases. However, most people can easily distinguish a computer graphics image from a photograph. Thus computer graphics have a 'realism' of their own that is a function of the model, and the nearness of the computer graphics image to a photograph of a real scene varies widely according to the method. Photo-realism in computer graphics means the image *looks* real not that it approaches, on a pixel by pixel basis, a photograph. This subjective judgement of computer graphics images somewhat devalues the widely used adjective 'photo-realistic', but there you are. With one or two exceptions very little work has been done on comparing a human's perception of a computer graphics image with, say, a TV image of the equivalent real scene.

Acknowledgements

The author would like to thank the following:

- Lightwork Design Ltd (Sheffield, UK) and Dave Cauldron for providing the facilities to produce the front cover image (model of the Tate Gallery, St Ives, UK) and the renderer, RadioRay.
- Daniel Teece for the images on the back cover which he produced as part of his PhD thesis and which comprise three-dimensional paint strokes interactively applied to a standard polygon model.
- Lee Cooper for producing Figures 6.12, 7.8, 8.7, 8.10, 10.4, 18.1, 18.3, 18.5, 18.6, 18.7, 18.8, 18.9, 18.10, 18.11, 18.12, 18.13, 18.14, 18.16, 18.17 and 18.19 together with the majority of images on the CD-ROM. These were produced using Lightworks Application Development System kindly supplied by Lightwork Design Ltd.
- Mark Puller for Figure 13.1.
- Steve Maddock for Figures 1.5, 4.9, 8.8, 8.26.
- Agata Opalach for Figure 2.20.
- Klaus de Geuss for Figures 13.10 and 13.11.
- Guy Brown for Figure 16.19.
- Fabio Policarpo for Figure 8.14.
- IMDM University, Hamburg, for Figure 13.3.

In addition the author would like to thank Keith Mansfield, the production staff at Addison-Wesley, Robert Chaundry of Bookstyle for his care with the manuscript and Dionea Watt for the cover design.

The publishers are grateful to the following for permission to reproduce copyright material:

Figure 2.1 reproduced with the permission of Viewpoint Digital, Inc; Figure 2.4 from *Tutorial: Computer Graphics*, Ze (Beatty and Booth, 1982), © 1982 IEEE, The Institute of Electrical and Electronics Engineers, Inc., New York; Figures 2.7 and 2.8 from *Generative Modelling for Computer Graphics and CAD* (Snyder, 1992), Academic

Press, London; Figure 2.20 reproduced with the permission of Agata Opalach; Figure 13.3 from *VOXEL-MAN, Part 1: Brain and Skull*, CD-ROM for UNIX workstations and LINUX PCs, Version 1.1 © Karl-Heinz Höhne and Springer-Verlag GmbH & Co. KG 1996, reproduced with kind permission; Figure 16.14 reproduced with the permission of Steven Seitz; Figure 17.28 from *ACM Transactions on Graphics*, 15:3, July 1996 (Hubbard, 1996), ACM Publications, New York.

Whilst every effort has been made to trace the owners of copyright material, in a few cases this has proved impossible and we take this opportunity to offer our apologies to any copyright holders whose rights we may have unwittingly infringed.

Trademark notice

1 Mathematical fundamentals of computer graphics

1.1 Manipulating three-dimensional structures

Transformations are important tools in generating three-dimensional scenes. They are used to move objects around in an environment, and also to construct a two-dimensional view of the environment for a display surface. This chapter deals with basic three-dimensional transformations, and introduces some useful shape-changing transformations and basic three-dimensional geometry that we will be using later in the text.

In computer graphics the most popular method for representing an object is the polygon mesh model. This representation is fully described in Chapter 2. We do this by representing the surface of an object as a set of connected planar polygons and each polygon is a list of (connected) points. This form of representation is either exact or an approximation depending on the nature of the object. A cube, for example, can be represented exactly by six squares. A cylinder, on the other hand can only be approximated by polygons; say six rectangles for the curved surface and two hexagons for the end faces. The number of polygons used in the approximation determines how accurately the object is represented and this has repercussions in modelling cost, storage and rendering cost and quality. The popularity of the polygon mesh modelling technique in computer graphics is undoubtedly due to its inherent simplicity and the development of inexpensive shading algorithms that work with such models.

$$p_a = \frac{1}{y_1 - y_2}\,[p_1(y_s - y_2) + p_2\,(y_1 - y_s)]$$

$$p_b = \frac{1}{y_1 - y_4}\,[p_1(y_s - y_4) + p_4\,(y_1 - y_s)]$$

$$p_s = \frac{1}{x_b - x_a}\,[p_a(x_b - x_s) + p_b\,(x_s - x_a)]$$

These would normally be implemented using an incremental form, the final equation, for example, becoming:

$$p_s := p_s + \Delta p$$

with the constant value Δp calculated once per scan line.

2 Representation and modelling of three-dimensional objects (1)

Introduction

The primary purpose of three-dimensional computer graphics is to produce a two-dimensional image of a scene or an object from a description or model of the object. The object may be a real or existing object or it may exist only as a computer description. A less common but extremely important usage is where the act of creation of the object model and the visualization are intertwined. This occurs in interactive CAD applications where a designer uses the visualization to assist the act of creating the object. Most object descriptions are approximate in the sense that they describe the geometry or shape of the object only to the extent that inputting this description to a renderer produces an image of acceptable quality. In many CAD applications, however, the description has to be accurate because it is used to drive a manufacturing process. The final output is not a two-dimensional image but a real three-dimensional object.

Modelling and representation is a general phrase which can be applied to any or all of the following aspects of objects:

- Creation of a three-dimensional computer graphics representation.
- The technique or method or data structure used to represent the object.

- Manipulation of the representation – in particular changing the shape of an existing model.

The ways in which we can create computer graphics objects are almost as many and varied as the objects themselves. For example, we might construct an architectural object through a CAD interface. We may take data directly from a device such as a laser ranger or a three-dimensional digitizer. We may use some interface based on a sweeping technique where so-called ducted solids are created by sweeping a cross-section along a spine curve. Creation methods have up to now tended to be manual or semi-manual involving a designer working with an interface. As the demand for the representation of highly complex scenes increases – from such applications as virtual reality (VR) – automatic methods are being investigated. For VR applications of existing realities the creation of computer graphics representations from photographs or video is an attractive proposition.

The representation of an object is very much an unsolved problem in computer graphics. We can distinguish between a representation that is required for a machine or renderer and the representation that is required by a user or user interface. Representing an object using polygonal facets – a polygon mesh representation – is the most popular machine representation. It is, however, an inconvenient representation for a user or creator of an object. Despite this it is used as both a user and a machine representation. Other methods have separate user and machine representations. For example, bi-cubic parametric patches and CSG methods, which constitute user or interface representations may be converted into polygon meshes for rendering.

The polygon mesh form suffers from many disadvantages when the object is complex and detailed. In mainstream computer graphics the number of polygons in an object representation can be anything from a few tens to hundreds of thousands. This has serious ramifications in rendering time and object creation cost and in the feasibility of using such objects in an animation or virtual reality environment. Other problems accrue in animation where a model has both to represent the shape of the object and be controlled by an animation system which may require collisions to be calculated or the object to change shape as a function of time. Despite this the polygon mesh is supreme in mainstream computer graphics. Its inertia is due in part to the development of efficient algorithms and hardware to render this description. This has resulted in a somewhat strange situation where it is more efficient – as far as rendering is concerned – to represent a shape with many simple elements (polygons) than to represent it with far fewer (and more accurate) but more complicated elements such as bi-cubic parametric patches (see Section 3.4.2).

The ability to manipulate the shape of an existing object depends strongly on the representation. Polygon meshes do not admit simple shape manipulation. Moving mesh vertices immediately disrupts the 'polygonal resolution' where a shape has been converted into polygons with some degree of accuracy that is related to the local curvature of the surface being represented. For example, imagine twisting a cube represented by six squares. The twisted object cannot be

represented by retaining only six polygons. Another problem with shape manipulation is scale. Sometimes we want to alter a large part of an object which may involve moving many elements at the same time; other times we may require a detailed change.

Different representational methods have their advantages and disadvantages but there is no universal solution to the many problems that still exist. Rather, particular modelling methods have evolved for particular contexts. A good example of this tendency is the development of constructive solid geometry methods (CSG) popular in interactive CAD because they facilitate an intuitive interface for the interactive design of complex industrial objects as well as a representation. CSG is a constrained representation in that we can only use it to model shapes that are made up of allowed combinations of the primitive shapes or elements that are included in the system.

How do we choose a representation? The answer is that it depends on the nature of the object, the particular computer graphics technique that we are going to use to bring the object to life and the application. All these factors are interrelated. We can represent some three-dimensional objects exactly using a mathematical formulation, for example, a cylinder or a sphere; for others we use an approximate representation. For objects that cannot be represented exactly by mathematics there is a trade-off between the accuracy of the representation and the bulk of information used. This is illustrated by the polygon mesh skeletons in Figure 2.1. You can only increase the veracity of the representation by increasing the polygonal resolution which then has high cost implications in rendering time.

The ultimate impossibility of this extrapolation has led to hybrid methods for very complex and unique objects such as a human head. For example, in representing a particular human head we can use a combination of a polygon mesh model and photographic texture maps. The solid form of the head is represented by a generic polygon mesh which is pulled around to match the actual dimensions of the head to be modelled. The detailed likeness is obtained by mapping a photographic texture onto this mesh. The idea here is that the detailed variations in the geometry are suggested by the texture map rather than by detailed excursions in the geometry. Of course, its not perfect because the detail in the photograph depends on the lighting conditions under which it was taken as well as the real geometric detail, but it is a trick that is increasingly being used. Whether we regard the texture mapping as part of the representation or as part of the rendering process is perhaps a matter of opinion; but certainly the use of photographic texture maps in this context enables us to represent a complex object like a human head with a small number of polygons plus a photograph.

This compromise between polygonal resolution and a photographic texture map can be taken to extremes. In the computer games industry the total number of polygons rendered to the screen must be within the limiting number that can be rendered at, say, 15 frames per second on a PC. A recent football game consists of players whose heads are modelled with just a cube onto which a photographic texture is mapped.

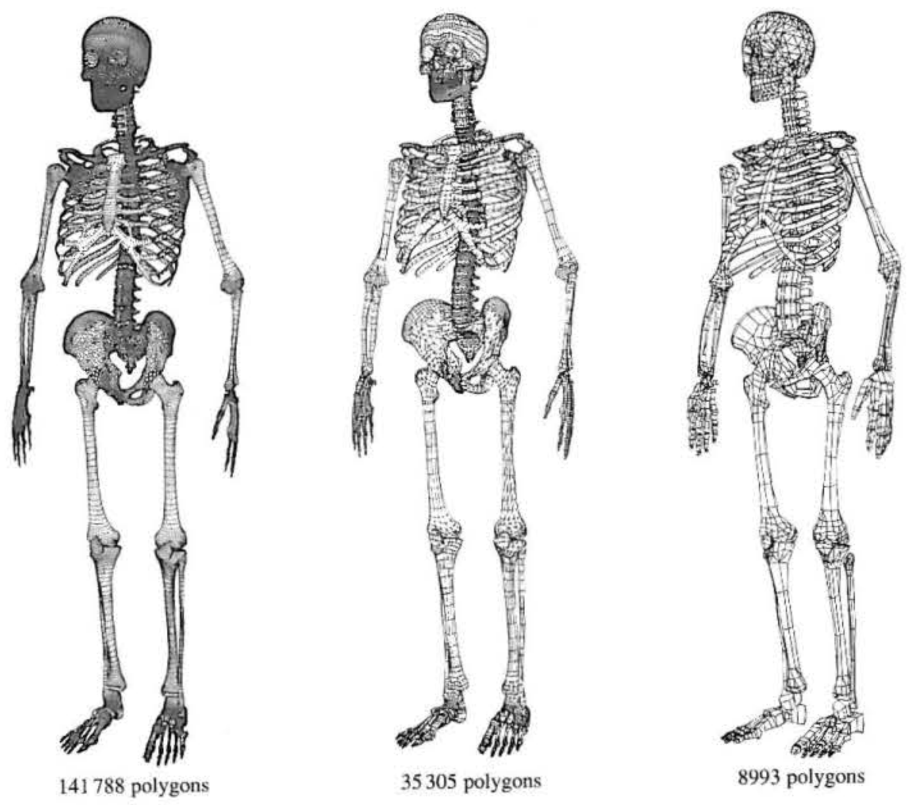


Figure 2.1
The art of wireframe – an illustration from Viewpoint Digital's catalogue.
Source: '3D models by Viewpoint Digital, Inc.' *Anatomy*, Viewpoint's 3D Dataset™ Catalog, 2nd edn.

We now list, in order of approximate frequency of use, the mainstream models used in computer graphics.

(1) **Polygonal** Objects are approximated by a net or mesh of planar polygonal facets. With this form we can represent, to an accuracy that we choose, an object of any shape. However, the accuracy is somewhat arbitrary in this sense. Consider Figure 2.1 again: are 142 000 polygons really necessary, or can we reduce the polygonal resolution without degrading the rendered image, and if so by how much? The shading algorithms are designed to visually transform the faceted representation in such a way that the piecewise linear representation is not visible in the shaded version (except on the silhouette edge). Connected with the polygonal resolution is the final projected size of the object on the screen. Waste is incurred when a complex object, represented by many thousands of polygons, projects onto a screen area that is made up of only a few pixels.

(2) **Bi-cubic parametric patches** (see Chapter 3) These are 'curved quadrilaterals'. Generally we can say that the representation is similar to the polygon mesh except that the individual polygons are now curved surfaces. Each patch is specified by a mathematical formula that gives the position of the patch in three-dimensional space and its shape. This formula enables us to generate any or every point on the surface of the patch. We can change the shape or curvature of the patch by editing the mathematical specification. This results in powerful interactive possibilities. The problems are, however, significant. It is very expensive to render or visualize the patches. When we change the shape of individual patches in a net of patches there are problems in maintaining 'smoothness' between the patch and its neighbours. Bi-cubic parametric patches can be either an exact or an approximate representation. They can only be an exact representation of themselves, which means that any object, say, a car body panel, can only be represented exactly if its shape corresponds exactly to the shape of the patch. This somewhat torturous statement is necessary because when the representation is used for real or existing objects, the shape modelled will not necessarily correspond to the surface of the object.

An example of the same object represented by both bi-cubic parametric patches and by polygonal facets is shown in Figure 3.28 (a) and (c). This clearly shows the complexity/number of elements trade-off with the polygon mesh representation requiring 2048 elements against the 32-patch representation.

(3) **Constructive solid geometry (CSG)** This is an exact representation to within certain rigid shape limits. It has arisen out of the realization that very many manufactured objects can be represented by 'combinations' of elementary shapes or geometric primitives. For example, a chunk of metal with a hole in it could be specified as the result of a three-dimensional subtraction between a rectangular solid and a cylinder. Connected with this is the fact that such a representation makes for easy and intuitive shape control – we can specify that a metal plate has to have a hole in it by defining a cylinder of appropriate radius and subtracting it from the rectangular solid, representing the plate. The CSG method is a volumetric representation – shape is represented by elementary volumes or primitives. This contrasts with the previous two methods which represent shape using surfaces. An example of a CSG-represented object is shown in Figure 2.14.

(4) **Spatial subdivision techniques** This simply means dividing the object space into elementary cubes, known as voxels, and labelling each voxel as empty or as containing part of an object. It is the three-dimensional analogue of representing a two-dimensional object as the collection of pixels onto which the object projects. Labelling all of three-dimensional object space in this way is clearly expensive, but it has found applications in computer graphics. In particular, in ray tracing where an efficient algorithm results if the objects are represented in this way. An example of a voxel object is shown in Figure 2.16. We are now representing the three-dimensional space occupied by the object; the other methods we have introduced are representations of the surface of the object.

(5) **Implicit representation** Occasionally in texts implicit functions are mentioned as an object representation form. An implicit function is, for example:

$$x^2 + y^2 + z^2 = r^2$$

which is the definition for a sphere. On their own these are of limited usefulness in computer graphics because there is a limited number of objects that can be represented in this way. Also, it is an inconvenient form as far as rendering is concerned. However, we should mention that such representations do appear quite frequently in three-dimensional computer graphics – in particular in ray tracing where spheres are used frequently – both as objects in their own right and as bounding objects for other polygon mesh representations.

Implicit representations are extended into implicit functions which can loosely be described as objects formed by mathematically defining a surface that is influenced by a collection of underlying primitives such as spheres. Implicit functions find their main use in shape-changing animation – they are of limited usefulness for representing real objects.

We have arranged the categories in order of popularity; another useful comparison is: with voxels and polygon meshes the number of representational elements per object is likely to be high (if accuracy is to be achieved) but the complexity of the representation is low. This contrasts with bi-cubic patches where the number of elements is likely to be much lower in most contexts but the complexity of the representation is higher.

We should not deduce from the above categorization that the choice of a representation is a free one. The representational form is decided by both the rendering technique and the application. Consider, for example, the continuous/discrete representation distinction. A discrete representation – the polygon mesh – is used to represent the arbitrary shapes of existing real world objects – it is difficult to see how else we would deal with such objects. In medical imaging the initial representation is discrete (voxels) because this is what the imaging technology produces. On the other hand in CAD work we need a continuous representation because eventually we are going to produce, say, a machine part from the internal description. The representation has, therefore, to be exact.

The CSG representation does not fit easily into these comparisons. It is both a discrete and a continuous representation, being a discrete combination of interacting primitives, some of which can be described by a continuous function.

Another important distinguishing factor is surface versus volume representation. The polygon mesh is an approximate representation of the surface of an object and the rendering engine is concerned with providing a visualization of that surface. With Gouraud shading the algorithm is only concerned with using geometric properties associated with the surface representation. In ray tracing, because the bulk of the cost is involved in tracking rays through space and finding which objects they intersect, a surface representation implies high rendering cost. Using a volume representation, where the object space is labelled according to object occupancy, greatly reduces the overall cost of rendering.

The relationship between a rendering method and the representation is critically important in the radiosity method and here, to avoid major defects in the final image, there has to be some kind of interaction between the representation and the execution of the algorithm. As the algorithm progresses the representation must adapt so that more accurate consideration is given to areas in the emerging solution that need greater consideration. In other words, because of the expense of the method, it is difficult to decide *a priori* what the level of detail in the representation should be. The unwieldiness of the concept of having a scene representation depend on the progress of the rendering algorithm is at the root of the difficulty of the radiosity method and is responsible for its (current) lack of uptake as a mainstream tool.

2.1 Polygonal representation of three-dimensional objects

This is the classic representational form in three-dimensional graphics. An object is represented by a mesh of polygonal facets. In the general case an object possesses curved surfaces and the facets are an approximation to such a surface (Figure 2.2). Polygons may contain a vertex count that emerges from the technology used to create the model, or we may constrain all polygons to be triangles. It may be necessary to do this, for example, to gain optimal performance from special-purpose hardware or graphics accelerator cards.

Polygonal representations are ubiquitous in computer graphics. There are two reasons for this. Creating polygonal objects is straightforward (although for complex objects the process can be time consuming and costly) and visually effective algorithms exist to produce shaded versions of objects represented in this way. As we have already stated, polygon meshes are strictly a machine representation – rather than a convenient user representation – and they often function in this capacity for other representations which are not directly renderable. Thus bi-cubic parametric patches, CSG and voxel representations are often converted into polygon meshes prior to rendering

There are certain practical difficulties with polygon meshes. Foremost amongst these is accuracy. The accuracy of the model, or the difference between the faceted representation and the curved surface of the object, is usually arbitrary. As far as final image quality is concerned, the size of individual polygons should ideally depend on local spatial curvature. Where the curvature changes

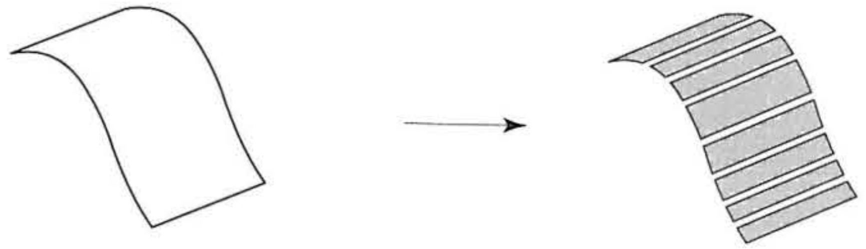


Figure 2.2
Approximating a curved surface using polygonal facets.

rapidly, more polygons are required per unit area of the surface. These factors tend to be related to the method used for creating the polygons. If, for example, a mesh is being built from an existing object, by using a three-dimensional digitizer to determine the spatial coordinates of polygon vertices, the digitizer operator will decide on the basis of experience how large each polygon should be. Sometimes polygons are extracted algorithmically (as in, for example, the creation of an object as a solid of revolution or in a bi-cubic patch subdivision algorithm) and a more rigorous approach to the rate of polygons per unit area of the surface is possible.

One of the most significant developments in three-dimensional graphics was the emergence in the 1970s of shading algorithms that deal efficiently with polygonal objects, and at the same time, through an interpolation scheme, diminish the visual effect of the piecewise linearities in the representation. This factor, together with recent developments in fixed program rendering hardware, has secured the entrenchment of the polygon mesh structure.

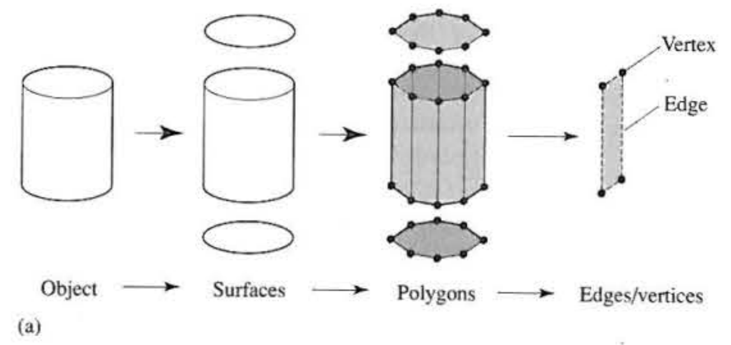
In the simplest case a polygon mesh is a structure that consists of polygons represented by a list of linked (x, y, z) coordinates that are the polygon vertices (edges are represented either explicitly or implicitly as we shall see in a moment). Thus the information we store to describe an object is finally a list of points or vertices. We may also store, as part of the object representation, other geometric information that is used in subsequent processing. These are usually polygon normals and vertex normals. Calculated once only, it is convenient to store these in the object data structure and have them undergo any linear transformations that are applied to the object.

It is convenient to order polygons into a simple hierarchical structure. Figure 2.3(a) shows a decomposition that we have called a conceptual hierarchy for reasons that should be apparent from the illustration. Polygons are grouped into surfaces and surfaces are grouped into objects. For example, a cylinder possesses three surfaces: a planar top and bottom surface together with a curved surface. The reason for this grouping is that we must distinguish between those edges that are part of the approximation – edges between adjacent rectangles in the curved surface approximation to the cylinder, for example – and edges that exist in reality. The way in which these are subsequently treated by the rendering process is different – real edges must remain visible whereas edges that form part of the approximation to a curved surface must be made invisible. Figure 2.3(b) shows a more formal representation of the topology in Figure 2.3(a).

An example of a practical data structure which implements these relationships is shown in Figure 2.3(c). This contains horizontal, as well as vertical, hierarchical links, necessary for programmer access to the next entity in a horizontal sequence. It also includes a vertex reference list which means that actual vertices (referred to by each polygon that shares them) are stored only once. Another difference between the practical structure and the topological diagram is that access is allowed directly to lower-level entities. Wireframe visualizations of an object are used extensively, and to produce a wireframe image requires direct access to the edge level in the hierarchy. Vertical links between the edges' and the

polygons' levels can be either backward pointers or forward pointers depending on the type of renderer that is accessing the structure. In a scan line renderer, edges are the topmost entity whereas in a Z-buffer renderer polygons are. A Z-buffer renderer treats polygons as independent entities, rendering one polygon at a time. A scan line renders all those polygons that straddle the scan line being rendered.

The approach just described is more particularly referred to as a vertex-based boundary model. Sometimes it is necessary to use an edge-based boundary



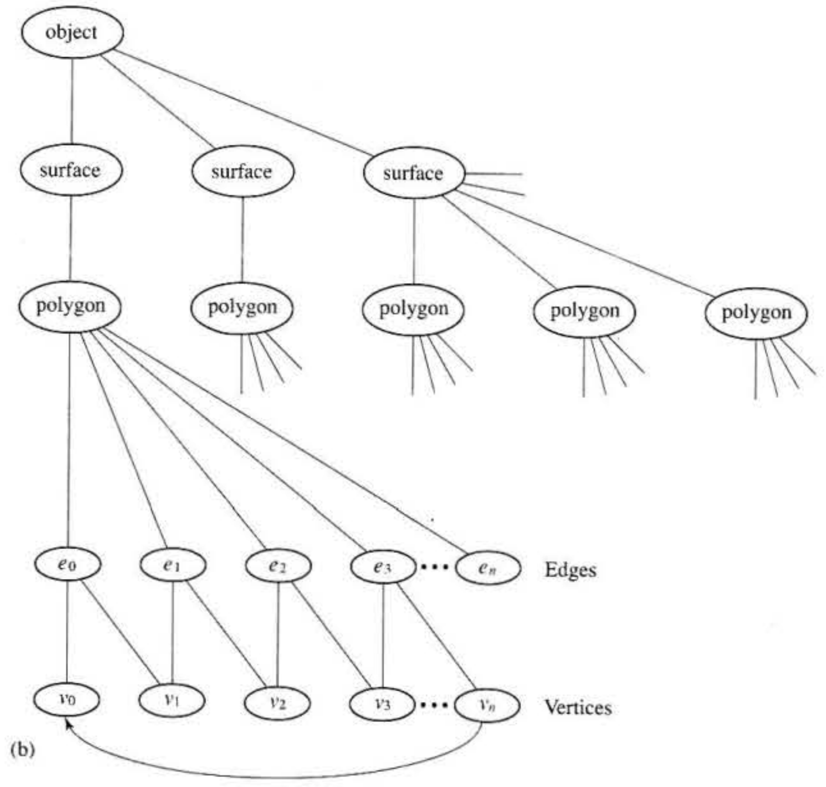
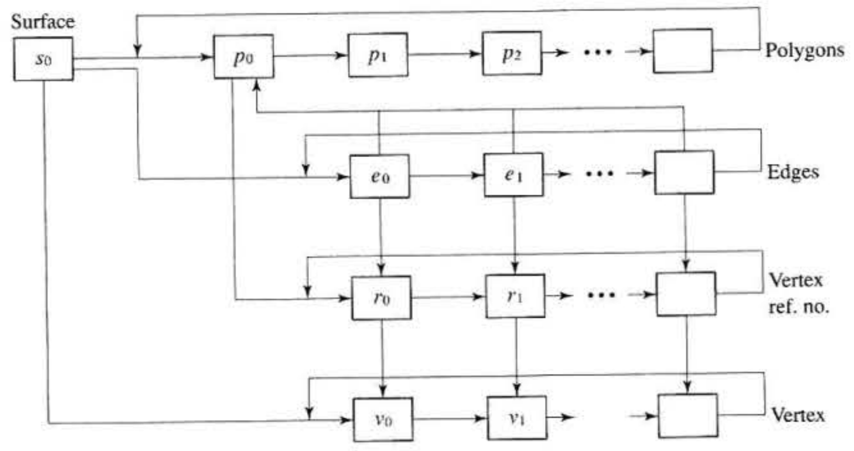


Figure 2.3
Representation of an object as a mesh of polygons.
(a) Conceptual hierarchy.
(b) Topological representation.



(c)

Figure 2.3 ***continued***
(c) A practical data structure.

model, the most common manifestation of which is a winged-edge data structure (Mantyla 1988). An edge-based model represents a face in terms of a closing sequence of edges.

The data structure just described encapsulates the basic geometry associated with a polygonal facets of an object. Information required by applications and renderers is also usually contained in the scene/object database. The following list details the most common attributes found in polygon mesh structures. They are either data structure pointers, real numbers or binary flags. It is unlikely that all of these would appear in a practical application, but a subset is found in most object representations.

- Polygon attributes
  - (1) Triangular or not.
  - (2) Area.
  - (3) Normal to the plane containing the polygon.
  - (4) Coefficients (A, B, C, D) of the plane containing the polygon where $Ax + By + Cz + D = 0$.
  - (5) Whether convex or not.
  - (6) Whether it contains holes or not.
- Edge attributes
  - (1) Length.
  - (2) Whether an edge is between two polygons or between two surfaces.
  - (3) Polygons on each side of the edge.
- Vertex attributes
  - (1) Polygons that contribute to the vertex.

  - (2) Shading or vertex normal – the average of the normals of the polygons that contribute to the vertex.
  - (3) Texture coordinates (u, v) specifying a mapping into a two-dimensional texture image.

All these are absolute propereties that exist when the object is created. Polygons can aquire attributes as they are passed through the graphics pipeline. For example, an edge can be tagged as a silhouette edge if it is between two polygons with normals facing towards and away from the viewer.

A significant problem that crops up in many guises in computer graphics is the scale problem. With polygonal representation this means that, in many applications, we cannot afford to render all the polygons in a model if the viewing distance and polygonal resolution are such that many polygons project onto a single pixel. This problem bedevils flight simulators (and similarly computer games) and virtual reality applications. An obvious solution is to have a hierarchy of models and use the one appropriate to projected screen area. There are two problems with this; the first is that in animation (and it is animation applications where this problem is most critical) switching between models can cause visual disturbances in the animation sequence – the user can see the switch from one resolution level to another. The other problem is how to generate the hierarchy and to decide how many levels it should contain. Clearly we can start with the highest resolution model and subdivide, but this is not necessarily straightforward. We look at this problem in more detail in Section 2.5.

2.1.1 Creating polygonal objects

Although a polygon mesh is the most common representational form in computer graphics, modelling, although straightforward, is somewhat tedious. The popularity of this representation derives from the ease of modelling, the emergence of rendering strategies (both hardware and software) to process polygonal objects and the important fact that there is no restriction whatever on the shape or complexity of the object being modelled.

Interactive development of a model is possible by 'pulling' vertices around with a three-dimensional locator device but in practice this is not a very useful method. It is difficult to make other than simple shape changes. Once an object has been created, any single polygon cannot be changed without also changing its neighbours. Thus most creation methods use either a device or a program; the only method that admits user interaction is item 4 on the following list.

Four common examples of polygon modelling methods are:

(1) Using a three-dimensional digitizer or adopting an equivalent manual strategy.
(2) Using an automatic device such as a laser ranger.
(3) Generating an object from a mathematical description.
(4) Generating an object by sweeping.

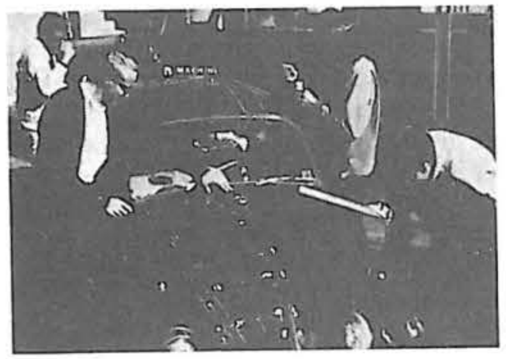
The first two modelling methods convert real objects into polygon meshes, the next two generate models from definitions. We distinguish between models generated by mathematical formulae and those generated by interacting with curves which are defined mathematically.

2.1.2 Manual modelling of polygonal objects

The easiest way to model a real object is manually using a three-dimensional digitizer. The operator uses experience and judgement to emplace points on an object which are to be polygon vertices. The three-dimensional coordinates of these vertices are then input to the system via a three-dimensional digitizer. The association of vertices with polygons is straightforward. A common strategy for ensuring an adequate representation is to draw a net over the surface of the object – like laying a real net over the object. Where curved net lines intersect defines the position of the polygon vertices. A historic photograph of this process is shown in Figure 2.4. This shows students creating a polygon mesh model of a car in 1974. It is taken from a classic paper by early outstanding pioneers in computer graphics – Sutherland *et al.* (1974).

2.1.3 Automatic generation of polygonal objects

A device that is capable of creating very accurate or high resolution polygon mesh objects from real objects is a laser ranger. In one type of device the object is placed on a rotating table in the path of the beam. The table also moves up and down vertically. The laser ranger returns a set of contours – the intersection of the object and a set of closely spaced parallel planes – by measuring the distance to the object surface. A 'skinning' algorithm, operating on pairs of contours, converts the boundary data into a very large number of triangles (Figure 2.5(a)). Figure 2.5(b) is a rendered version of an object polygonized in this way. The skinning algorithm produced, for this object, over 400 000 triangles. Given that only around half of these may be visible on screen and that the object



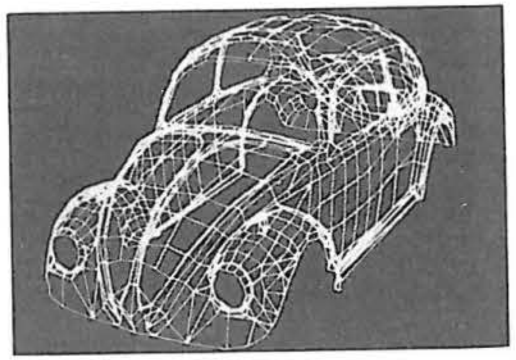
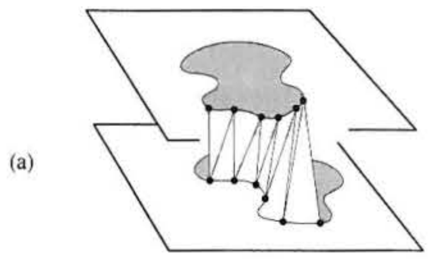


Figure 2.4
The Utah Beetle – an early example of manual modelling.
Source: Beatty and Booth *Tutorial: Computer Graphics*, 2nd edn, The Institute of Electrical and Electronics Engineers, Inc.: New York. © 1982 IEEE.



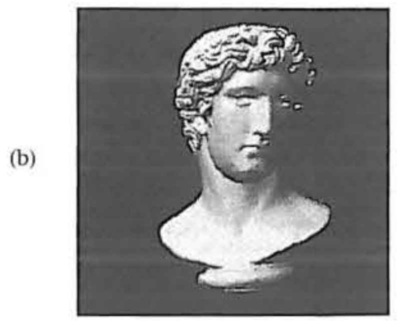


Figure 2.5
A rendered polygonal object scanned by a laser ranger and polygonized by a simple skinning algorithm.
(a) A skinning algorithm joins points on consecutive contours to make a three-dimensional polygonal object from the contours.
(b) A 400 000 polygonal object produced by a skinning algorithm.

projects onto about half the screen surface implies that each triangle projects onto one pixel on average. This clearly illustrates the point mentioned earlier that it is extremely wasteful of rendering resources to use a polygonal resolution where the average screen area onto which a polygon projects approaches a single pixel. For model creation, laser rangers suffer from the significant disadvantage that, in the framework described – fully automatic rotating table device – they can only accurately model convex objects. Objects with concavities will have surfaces which will not necessarily be hit by the incident beam.

2.1.4 Mathematical generation of polygonal objects

Many polygonal objects are generated through an interface into which a user puts a model description in the form of a set of curves that are a function of two-dimensional or two-parameter space. This is particularly the case in CAD applications where the most popular paradigm is that of sweeping a cross-section in a variety of different ways. There are two benefits to this approach. The first is fairly obvious. The user works with some notion of shape which is removed from the low level activity of constructing an object from individual polygonal facets. Instead, shape is specified in terms of notions that are connected with the form of the object – something that Snyder (1992) calls 'the logic of shapes'. A program then takes the user description and transforms it into polygons. The transformation from the user description to a polygon mesh is straightforward. A second advantage of this approach is that it can be used in conjunction with either polygons as primitive elements or with bi-cubic parametric patches (see Section 3.6).

The most familiar manifestation of this approach is a solid of revolution where, say, a vertical cross-section is swept through 180° generating a solid with a circular horizontal cross-section (Figure 2.6(a)). The obvious constraint of solids of revolution is that they can only represent objects possessing rotational symmetry.

A more powerful generative model is arrived at by considering the same solid generated by sweeping a circle, with radius controlled by a profile curve,

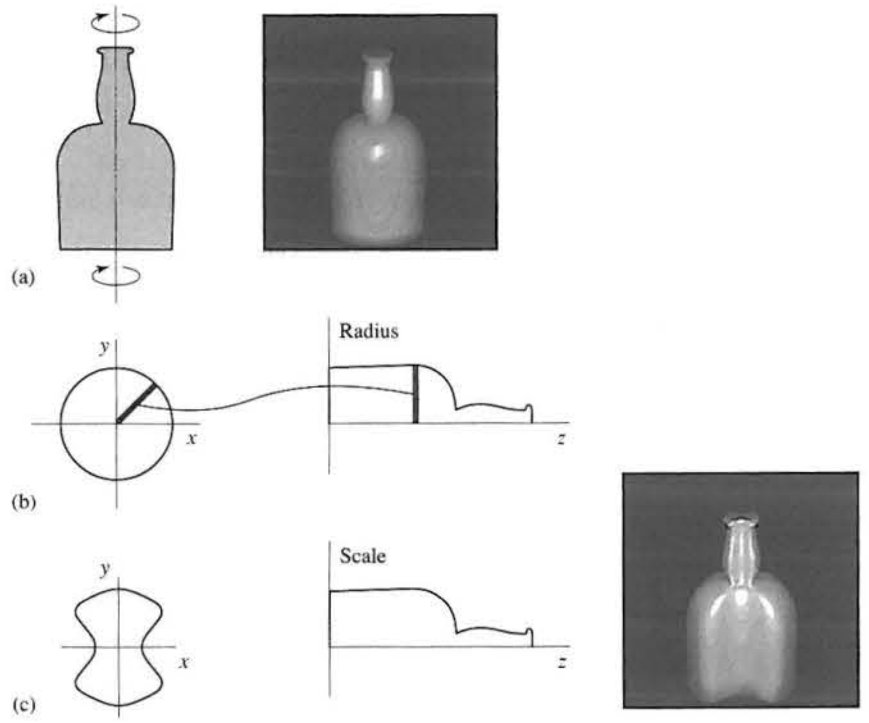


Figure 2.6
Straight spine objects – solid of revolution vs cross-sectional sweeping. (a) A solid of revolution generated by sweeping a (vertical) cross-section. (b) The same solid can be generated by sweeping a circle, whose radius is controlled by a profile curve, up a straight vertical spine. (c) Non-circular cross-section.

vertically up a straight spine (Figure 2.6(b)). In the event that the profile curve is a constant, we have the familiar notion of extrusion. This immediately removes the constraint of a circular cross-section and we can have cross-sections of arbitrary shape (Figure 2.6(c)).

Now consider controlling the shape of the spine. We can incorporate the notion of a curved spine and generate objects that are controlled by a cross-sectional shape, a profile curve and a spine curve as Figure 2.9 demonstrates.

Other possibilities emerge. Figure 2.7 shows an example of what Snyder calls a rail product surface. Here a briefcase carrying handle is generated by sweeping a cross-section along a path determined by the midpoints of two rail curves. The long axis extent of the elliptical-like cross-section is controlled by the same two curves – hence the name. A more complex example is the turbine blade shown in Figure 2.8. Snyder calls this an affine transformation surface – because the spine is now replaced by affine transformations, controlled by user specified curves. Each blade is generated by extruding a rectangular cross-section along the z axis. The cross-section is specified as a rectangle, and three shape controlling curves, functions of z, supply the values used in the transformations of the cross-section as it is extruded. The cross-section is, for each step in z, scaled separately in x and y, translated in x, rotated around, translated back in x, and extruded along the z axis.

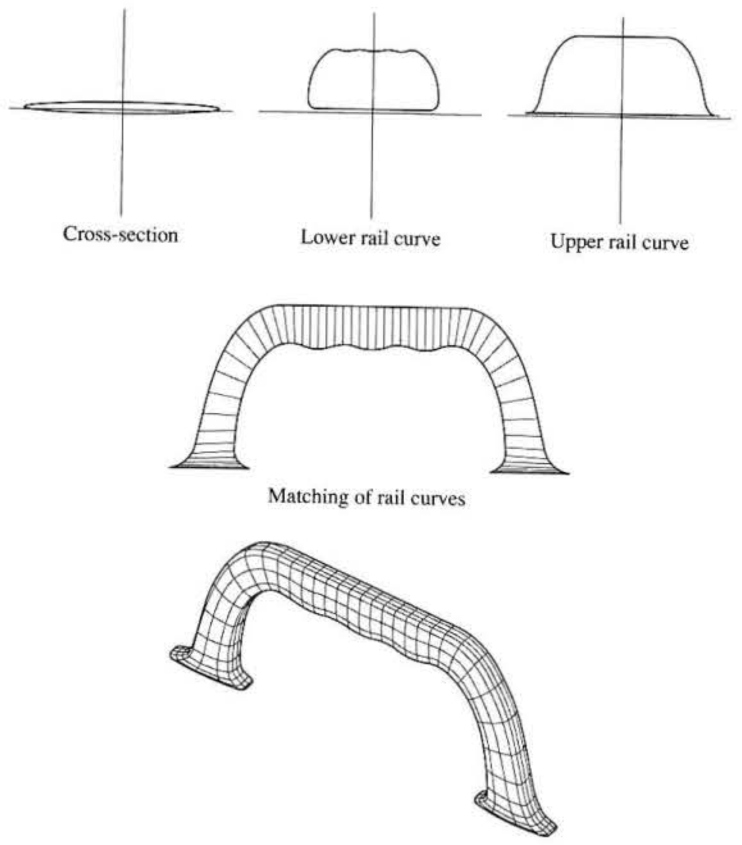


Figure 2.7
Snyder's rail curve product surfaces. *Source:* J.M. Snyder, *Generative Modelling for Computer Graphics and CAD*, Academic Press, 1992.

A complicated shape is thus generated by a general cross-section and three curves. Clearly implicit in this example is a reliance on a user/designer being able to visualize the final required shape in three-dimensions so that he is able to specify the appropriate shape curves. Although for the turbine blade example this may seem a somewhat tall order, we should bear in mind that shapes of such complexity are the domain of professional engineers where the use of such generative models for shape specification will not be unfamiliar.

Certain practical problems emerge when we generalize to curved spines. There are three difficulties in allowing curved spines that immediately emerge. These are illustrated in Figure 2.9. Figure 2.9(a) shows a problem in the curve to polygon procedure. Here it is seen that the size of the polygonal primitives depends on the excursion of the spine curve. The other is how do we orient the cross-section with respect to a varying spine (Figure 2.9(b))? And, finally, how do we prevent cross-sections self-intersecting (Figure 2.9(c))? It is clear that this will occur as soon as the radius of curvature of the path of any points traced out by the cross-sectional curve exceeds the radius of curvature of the spine. We will now look at approaches to these problems.

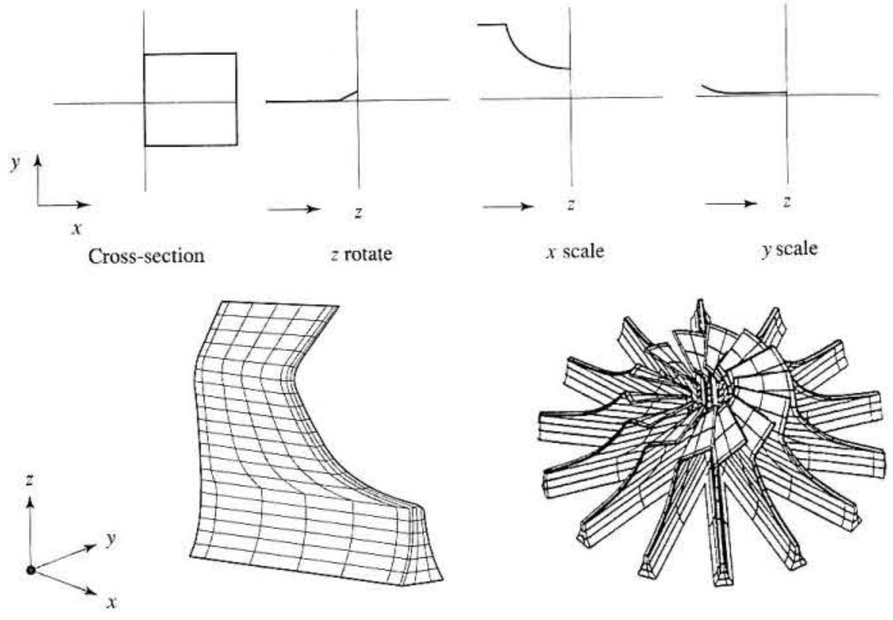


Figure 2.8
Snyder's affine transformation surface. The generating curves are shown for a single turbine blade. *Source*: J.M. Snyder, *Generative Modelling for Computer Graphics and CAD*, Academic Press, 1992.

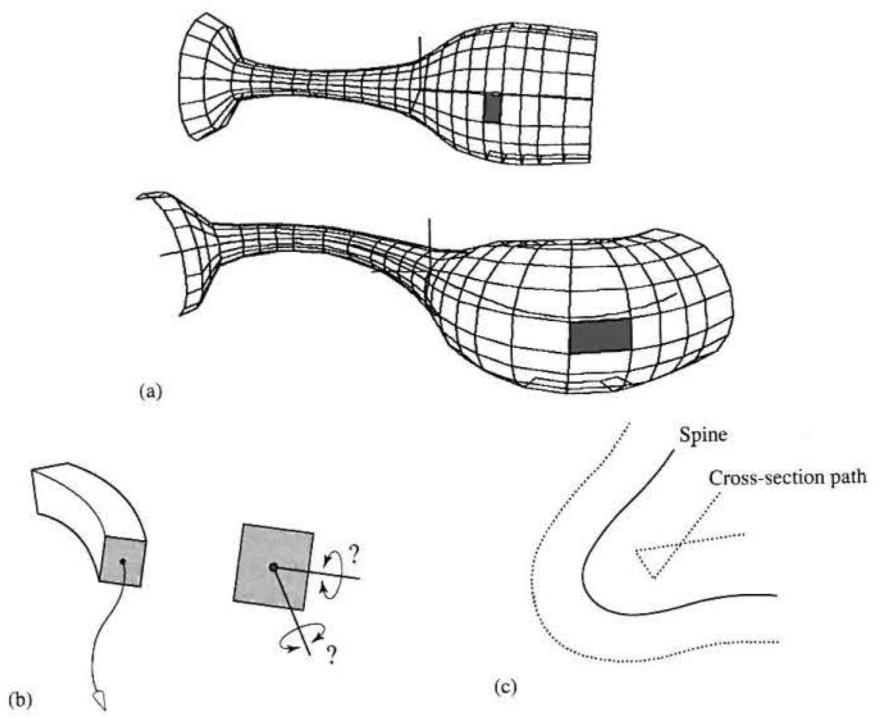


Figure 2.9
Three problems in cross-sectional sweeping.
(a) Controlling the size of the polygons can become problematic. (b) How should the cross-section be oriented with resepct to the spine curve? (c) Self-intersection of the cross-section path.

Consider a parametrically defined cubic along which the cross-section is swept. This can be defined (see Section 3.1) as:

$$\boldsymbol{Q}(u) = \boldsymbol{a}u^3 + \boldsymbol{b}u^2 + \boldsymbol{c}u + \boldsymbol{d}$$

Now if we consider the simple case of moving a constant cross-section without twisting it along the curve we need to define intervals along the curve at which the cross-section is to be placed and intervals around the cross-section curve. When we have these we can step along the spine intervals and around the cross-section intervals and output the polygons.

Consider the first problem. Dividing u into equal intervals will not necessarily give the best results. In particular the points will not appear at equal intervals along the curve. A procedure known as arc length parametrization divides the curve into equal intervals, but this procedure is not straightforward. Arc length parametrization may also be inappropriate. What is really required is a scheme that divides the curve into intervals that depend on the curvature of the curve. When the curvature is high the rate of polygon generation needs to be increased so that more polygons occur when the curvature twists rapidly. The most direct way to do this is to use the curve subdivision algorithm (see Section 4.2.3) and subdivide the curve until a linearity test is positive.

Now consider the second problem. Having defined a set of sample points we need to define a reference frame or coordinate system at each. The cross-section is then embedded in this coordinate system. This is done by deriving three mutually orthogonal vectors that form the coordinate axes. There are many possibilities.

A common one is the Frenet frame. The Frenet frame is defined by the origin or sample point, $\boldsymbol{P}$, and three vectors $\boldsymbol{T}$, $\boldsymbol{N}$ and $\boldsymbol{B}$ (Figure 2.10). $\boldsymbol{T}$ is the unit length tangent vector:

$$\boldsymbol{T} = \boldsymbol{V}/|\boldsymbol{V}|$$

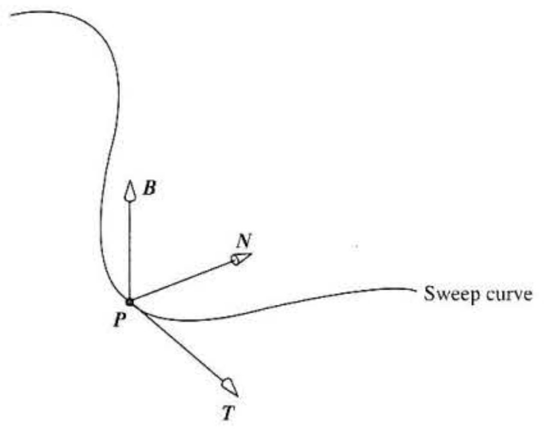


Figure 2.10
The Frenet frame at sample point ***P*** on a sweep curve.

where $\boldsymbol{V}$ is the derivative of the curve:

$$\boldsymbol{V} = 3\boldsymbol{a}u^2 + 2\boldsymbol{b}u + \mathbf{c}$$

The principal normal $\boldsymbol{N}$ is given by:

$$\boldsymbol{N} = \boldsymbol{K}/|\boldsymbol{K}|$$

where:

$$\boldsymbol{K} = \boldsymbol{V} \times \boldsymbol{A} \times \boldsymbol{V}/|\boldsymbol{V}|^4$$

and $\boldsymbol{A}$ is the second derivative of the curve:

$$\boldsymbol{A} = 6\boldsymbol{a}u + 2\boldsymbol{b}$$

Finally $\boldsymbol{B}$ is given by:

$$\boldsymbol{B} = \boldsymbol{T} \times \boldsymbol{N}$$

2.1.5 Procedural polygon mesh objects – fractal objects

In this section we will look at a common example of generating polygon mesh objects procedurally. Fractal geometry is a term coined by Benoit Mandelbrot (1977; 1982). The term was used to describe the attributes of certain natural phenomena, for example, coastlines. A coastline viewed at any level of detail – at microscopic level, at a level where individual rocks can be seen or at 'geographical' level, tends to exhibit the same level of jaggedness; a kind of statistical self-similarity. Fractal geometry provides a description for certain aspects of this ubiquitous phenomenon in nature and its tendency towards self-similarity.

In three-dimensional computer graphics, fractal techniques have commonly been used to generate terrain models and the easiest techniques involve subdividing the facets of the objects that consist of triangles or quadrilaterals. A recursive subdivision procedure is applied to each facet, to a required depth or level of detail, and a convincing terrain model results. Subdivision in this context means taking the midpoint along the edge between two vertices and perturbing it along a line normal to the edge. The result of this is to subdivide the original facets into a large number of smaller facets, each having a random orientation in three-dimensional space about the original facet orientation. The initial global shape of the object is retained to an extent that depends on the perturbation at the subdivision and a planar four-sided pyramid might turn into a 'Mont Blanc' shaped object.

Most subdivision algorithms are based on a formulation by Fournier *et al.* (1982) that recursively subdivides a single line segment. This algorithm was developed as an alternative to more mathematically correct but expensive procedures suggested by Mandelbrot. It uses self-similarity and conditional expectation properties of fractional Brownian motion to give an estimate of the increment of the stochastic process. The process is also Gaussian and the only parameters needed to describe a Gaussian distribution are the mean (conditional expectation) and the variance.

A procedure recursively subdivides a line (t_1, f_1), (t_2, f_2) generating a scalar displacement of the midpoint of the line in a direction normal to the line (Figure 2.11(a)).

To extend this procedure to, say, triangles or quadrilaterals in three-dimensional space, we treat each edge in turn generating a displacement along a midpoint vector that is normal to the plane of the original facet (Figure 2.11(b)). Using this technique we can take a smooth pyramid, say, made of large triangular faces and turn it into a rugged mountain.

Fournier categorizes two problems in this method – as internal and external consistency. Internal consistency requires that the shape generated should be the same whatever the orientation in which it is generated, and that coarser

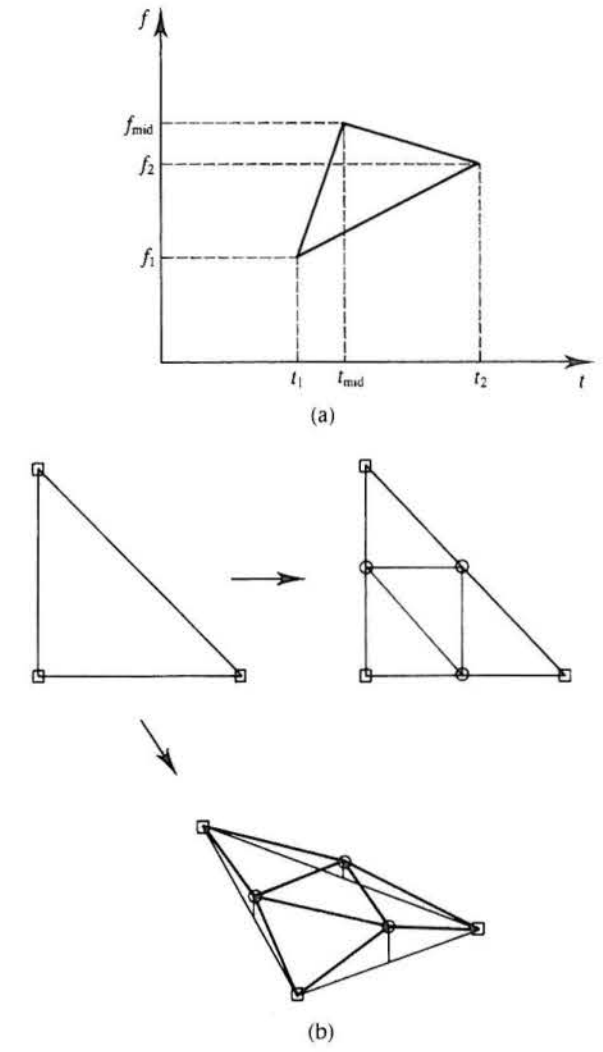


Figure 2.11
An example of procedural generation of polygon mesh objects – fractal terrain. (a) Line segment subdivision. (b) Triangle subdivision.

details should remain the same if the shape is replotted at greater resolution. To satisfy the first requirement, the Gaussian randoms generated must not be a function of the position of the points, but should be unique to the point itself. An invariant point identifier needs to be associated with each point. This problem can be solved in terrain generation by giving each point a key value used to index a Gaussian random number. A hash function can be used to map the two keys of the end points of a line to a key value for the midpoint. Scale requirements of internal consistency means that the same random numbers must always be generated in the same order at a given level of subdivision.

External consistency is harder to maintain. Within the mesh of triangles every triangle shares each of its sides with another; thus the same random displacements must be generated for corresponding points of different connecting triangles. This is already solved by using the key value of each point and the hash function, but another problem still exists, that of the direction of the displacement.

If the displacements are along the surface normal of the polygon under consideration, then adjacent polygons which have different normals (as is, by definition, always the case) will have their midpoints displaced into different positions. This causes gaps to open up. A solution is to displace the midpoint along the average of the normals to all the polygons that contain it but this problem occurs at every level of recursion and is consequently very expensive to implement. Also, this technique would create an unsatisfactory skyline because the displacements are not constrained to one direction. A better skyline is obtained by making all the displacements of points internal to the original polygon in a direction normal to the plane of the original polygon. This cheaper technique solves all problems relating to different surface normals, and the gaps created by them. Now surface normals need not be created at each level of recursion and the algorithm is considerably cheaper because of this.

Another two points are worth mentioning. Firstly, note that polygons should be constant shaded without calculating vertex normals – discontinuities between polygons should not be smoothed out. Secondly, consider colour. The usual global colour scheme uses a height-dependent mapping. In detail, the colour assigned to a midpoint is one of its end point's colours. The colour chosen is determined by a Boolean random which is indexed by the key value of the midpoint. Once again this must be accessed in this way to maintain consistency, which is just as important for colour as it is for position.

2.2 Constructive solid geometry (CSG) representation of objects

We categorized the previous method – polygon mesh – as a machine representation which also frequently functions as a user representation. The CSG approach is very much a user representation and requires special rendering techniques or the conversion to a polygon mesh model prior to representation. It is a high-level representation that functions both as a shape representation and a record

of how it was built up. The 'logic of the shape' in this representation is in how the final shape can be made or represented as a combination of primitive shapes. The designer builds up a shape by using the metaphor of three-dimensional building blocks and a selection of ways in which they can be combined. The high-level nature of the representation imposes a certain burden on the designer. Although with hindsight the logic of the parts in Figure 2.14 is apparent; the design of complex machine parts using this methodology is a demanding occupation.

The motivation for this type of representation is to facilitate an interactive mode for solid modelling. The idea is that objects are usually parts that will eventually be manufactured by casting, machining or extruding and they can be built up in a CAD program by using the equivalent (abstract) operations combining simple elementary objects called geometric primitives. These primitives are, for example, spheres, cones, cylinders or rectangular solids and they are combined using (three-dimensional) Boolean set operators and linear transformations. An object representation is stored as an attributed tree. The leaves contain simple primitives and the nodes store operators or linear transformations. The representation defines not only the shape of the object but its modelling history – the creation of the object and its representation become one and the same thing. The object is built up by adding primitives and causing them to combine with existing primitives. Shapes can be added to and subtracted from (to make holes) the current shape. For example, increasing the diameter of a hole through a rectangular solid means a trivial alteration – the radius of the cylinder primitive defining the hole is simply increased. This contrasts with the polygon mesh representation where the same operation is distinctly non-trivial. Even although the constituent polygons of the cylindrical surface are easily accessible in a hierarchical scheme, to generate a new set of polygons means reactivating whatever modelling procedure was used to create the original polygons. Also, account has to be taken of the fact that to maintain the same accuracy more polygons will have to be used.

Boolean set operators are used both as a representational form and as a user interface technique. A user specifies primitive solids and combines these using the Boolean set operators. The representation of the object is a reflection or recording of the user interaction operations. Thus we can say that the modelling information and representation are not separate – as they are in the case of deriving a representation from low-level information from an input device. The low-level information in the case of CSG is already in the form of volumetric primitives. The modelling activity becomes the representation. An example will demonstrate the idea.

Figure 2.12 shows the Boolean operations possible between solids. Figure 2.12(a) shows the union of two solids. If we consider the objects as 'clouds' of points the union operation encloses all points lying within the original two bodies. The second example (Figure 2.12(b)) shows the effect of a difference or subtraction operator. A subtract operator removes all those points in the second body that are contained within the first. In this case a cylinder is defined and

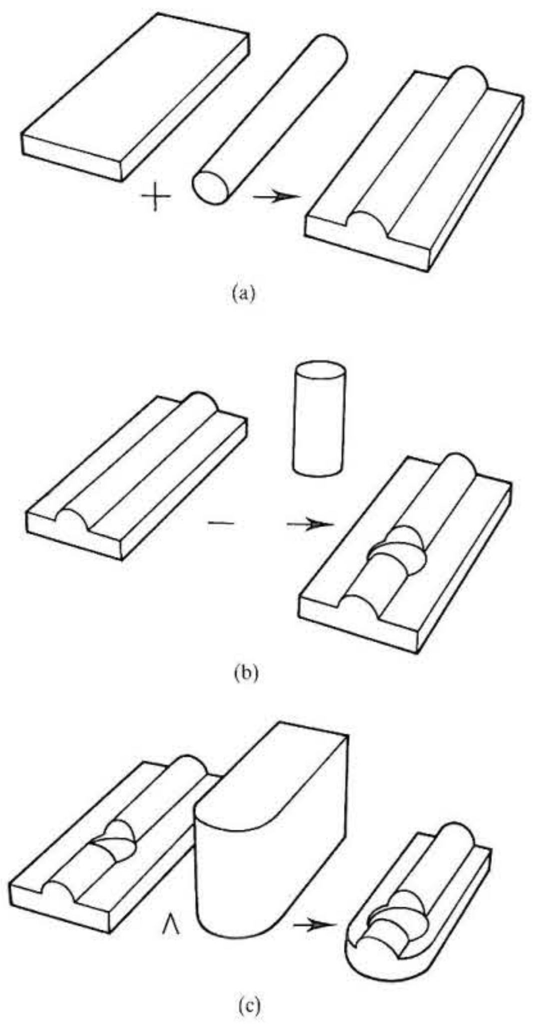


Figure 2.12
Boolean operations between solids in CSG modelling: (a) union, (b) subtraction and (c) intersection.

subtracted from the object produced in Figure 2.12(a). Finally, an example is shown of an intersect operation (Figure 2.12(c)). Here a solid is defined that is made from the union of a cylinder and a rectangular solid (the same operation with the same primitives as in Figure 2.12(a)). This solid then intersects with the object produced in Figure 2.12(b). An intersect operation produces a set of points that are contained by both the bodies. An obvious distinguishing feature of this method that follows from this example is that primitives are used not just to build up a model but also to take material away.

Figure 2.13 shows a CSG representation that reflects the construction of a simple object. Three original solids appear at the leaves of the tree: two boxes and a cylinder. The boxes are combined using a union operation and a hole is 'drilled' in one of the boxes by defining a cylinder and subtracting it from the two box

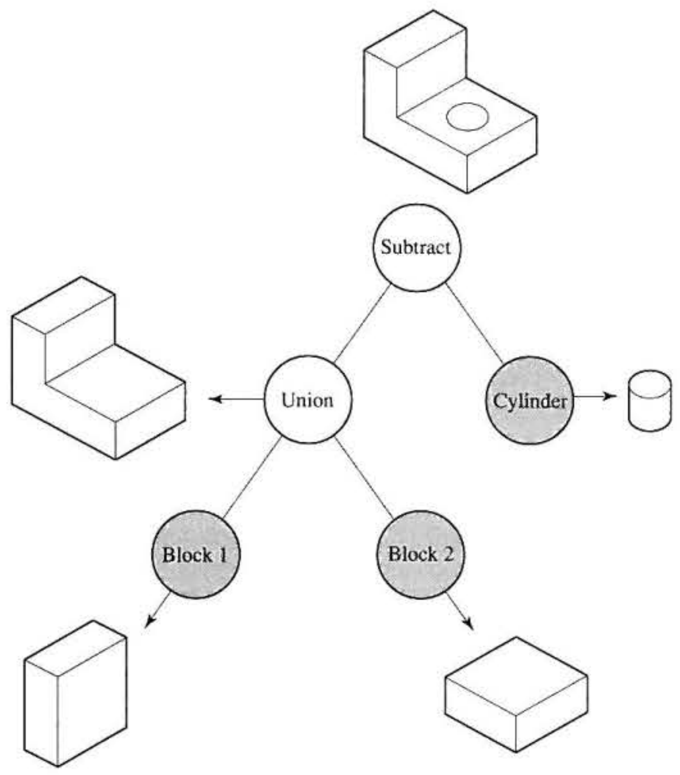


Figure 2.13
A CSG tree reflecting the construction of a simple object made from three primitives.

assembly. Thus the only information that has to be stored in the leaves of the tree is the name of the primitive and its dimensions. A node has to contain the name of the operator and the spatial relationship between the child nodes combined by the operator.

The power of Boolean operations is further demonstrated in the following examples. In the first example (Figure 2.14(a)) two parts developed separately are combined to make the desired configuration by using the union operator followed by a difference operator. The second example (Figure 2.14(b)) shows a complex object constructed only from the union of cylinders, which is then used to produce, by subtraction, a complex housing.

Although there are substantial advantages in CSG representation, they do suffer from drawbacks. A practical problem is the computation time required to produce a rendered image of the model. A more serious drawback is that the method imposes limitations on the operations available to create and modify a solid. Boolean operations are global – they affect the whole solid. Local operations, say a detailed modification on one face of a complex object cannot be easily implemented by using set operations. An important local modification required in many objects that are to be designed is blending surfaces. For example, consider the end face of a cylinder joined onto a flat base. Normally for practical manufacturing or aesthetic reasons, instead of the join being a right angle in cross-

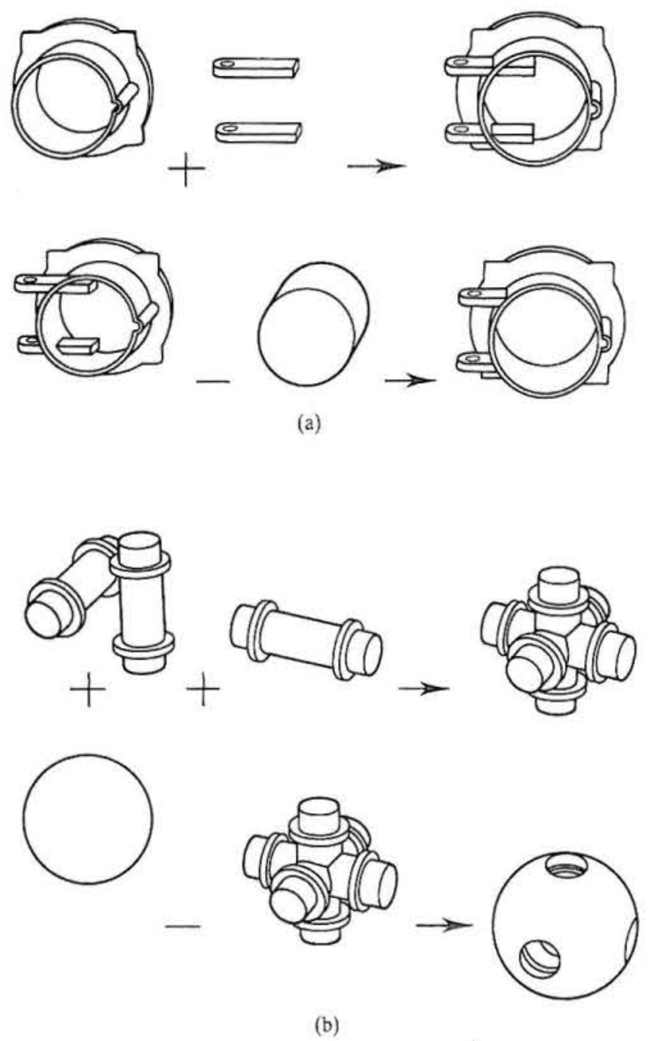


Figure 2.14
Examples of geometrically complex objects produced from simple objects and Boolean operations.

section a radius is desired. A radius swept around another curve cannot be represented in a simple CSG system. This fact has led to many solid modellers using an underlying boundary representation. Incidentally there is no reason why Boolean operations cannot be incorporated in boundary representations systems. For example, many systems incorporate Boolean operations but use a boundary representation to represent the object. The trade-off between these two representations has resulted in a debate that has lasted for 15 years. Finally note that a CSG representation is a volumetric representation. The space occupied by the object – its volume – is represented rather than the object surface.

2.3 Space subdivision techniques for object representation

Space subdivision techniques are methods that consider the whole of object space and in some way label each point in the space according to object occupancy. However, unlike CSG, which uses a variety of volumetric elements or geometric primitives, space subdivision techniques are based on a single cubic element known as a voxel. A voxel is a volumetric element or primitive and is the smallest cube used in the representation. We could divide up all of world space into regular or cubic voxels and label each voxel according to whether it is in the object or in empty space. Clearly this is very costly in terms of memory consumption. Because of this voxel representation is not usually a preferred mainstream method but is used either because the raw data are already in this form or it is easiest to convert the data into this representation – the case, for example, in medical imagery; or because of the demands of an algorithm. For example, ray tracing in voxel space has significant advantages over conventional ray tracing. This is an example of an algorithmic technique dictating the nature of the object representation. Here, instead of asking the question: 'does this ray intersect with any objects in the scene?' which implies a very expensive intersection test to be carried out on each object, we pose the question: 'what objects are encountered as we track a ray through voxel space?' This requires no exhaustive search through the primary data structure for possible intersections and is a much faster strategy.

Another example is rendering CSG models (Section 4.3) which is not straightforward if conventional techniques are used. A strategy is to convert the CSG tree into an intermediate data consisting of voxels and render from this. Voxels can be considered as an intermediate representation, most commonly in medical imaging where their use links two-dimensional raw data with the visualization of three-dimensional structures. Alternatively the raw data may themselves be voxels. This is the case with many mathematical modelling schemes of three-dimensional physical phenomena such as fluid dynamics.

The main problem with voxel labelling is the trade-off between the consumption of vast storage costs and accuracy. Consider, for example, labelling square pixels to represent a circle in two-dimensional space. The pixel size /accuracy trade-off is clear here. The same notion extends to using voxels to represent a sphere except that now the cost depends on the accuracy and the cube of the radius. Thus such schemes are only used in contexts where their advantages outweigh their cost. A way to reduce cost is to impose a structural organization on the basic voxel labelling scheme.

The common way of organizing voxel data is to use an octree – a hierarchical data structure that describes how the objects in a scene are distributed throughout the three-dimensional space occupied by the scene. The basic idea is shown in Figure 2.15. In Figure 2.15(a) a cubic space is subject to a recursive subdivision which enables any cubic region of the space to be labelled with a number. This subdivision can proceed to any desired level of accuracy. Figure 2.15(b) shows an object embedded in this space and Figure 2.15(c) shows the subdivision and the

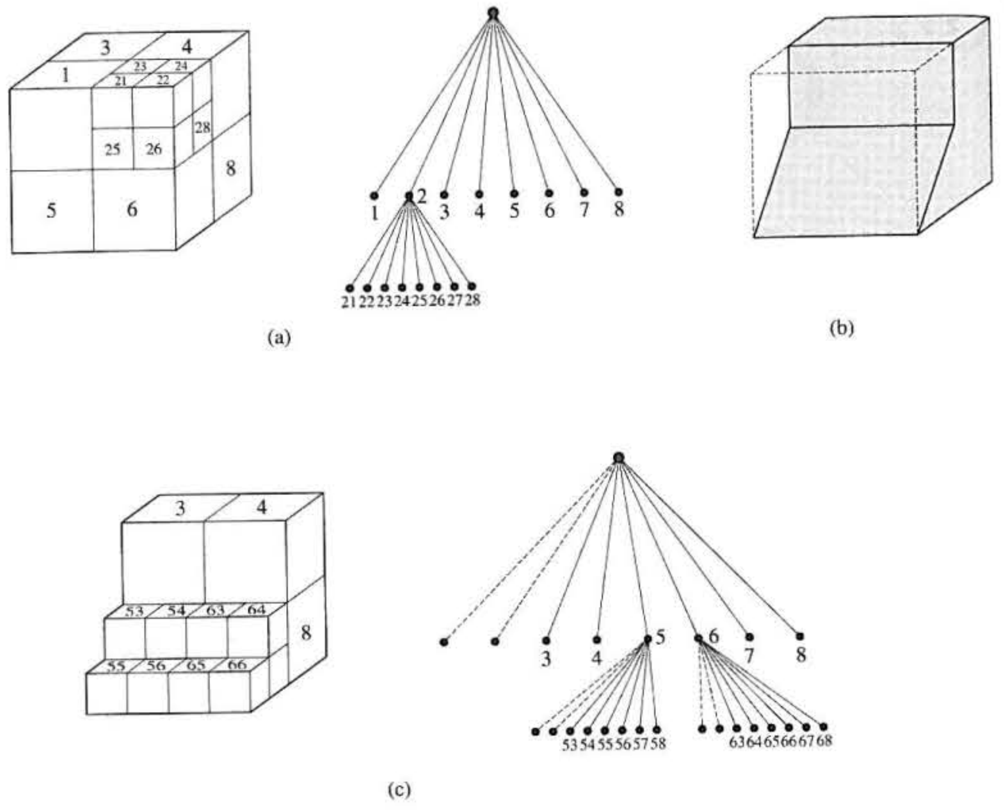


Figure 2.15
Octree representation. (a) Cubic space and labelling scheme, and the octree for the two levels of subdivision. (b) Object embedded in space. (c) Representation of the object to two levels of subdivision.

related octree that labels cubic regions in the space according to whether they are occupied or empty.

There are actually two ways in which the octree decomposition of a scene can be used to represent the scene. Firstly, an octree as described above can be used in itself as a complete representation of the objects in the scene. The set of cells occupied by an object constitute the representation of the object. However, for a complex scene, high resolution work would require the decomposition of occupied space into an extremely large number of cells and this technique requires enormous amounts of data storage. A common alternative is to use a standard data structure representation of the objects and to use the octree as a representation of the distribution of the objects in the scene. In this case, a terminal node of a tree representing an occupied region would be represented by a pointer to the data structure for any object (or part of an object) contained within that region. Figure 2.16 illustrates this possibility in the two-dimensional case. Here the region subdivision has stopped as soon as a region is encountered that intersects only one object. A region represented by a terminal node is not necessarily completely occupied by the object associated with that region. The shape of the object within the region would be described by its data structure representation. In the case of a surface model representation of a scene, the 'objects' would be polygons or patches. In general, an occupied region represented by a terminal node would intersect with several polygons and would be represented by a list of pointers into the object data structures. Thus unlike the other techniques that we have described octrees are generally not self-contained representational methods. They are instead usually part of a hybrid scheme.

2.3.1 Octrees and polygons

As we have already implied, the most common use of octrees in computer graphics is not to impose a data structure, on voxel data, but to organize a scene containing many objects (each of which is made up of many polygons) into a structure of spatial occupancy. We are not representing the objects using voxels, but considering the rectangular space occupied as polygons as entities which are represented by voxel space. As far as rendering is concerned we enclose parts of the scene, at some level of detail, in rectangular regions in the sense of Figure 2.16. For example, we may include groups of objects, single objects, parts of objects or even single polygons in an octree leaf node. This can greatly speed up many aspects of rendering and many rendering methods, particularly ray tracing as we have already suggested.

We will now use ray tracing as a particular example. The high inherent cost in naive ray tracing resides in intersection testing. As we follow a ray through the scene we have to find out if it collides with any object in the scene (and what the position of that point is). In the case that each ray is tested against all objects in the scene, where each object test implies testing against each polygon in the object, the rendering time, for scenes of reasonable complexity, becomes unacceptably high. If the scene is decomposed into an octree representation, then tracing a ray means tracking, using an incremental algorithm from voxel to voxel. Each voxel contains pointers to polygons that it contains and the ray is tested against these. Intersection candidates are reduced from n to m, where:

$$n = \sum_{\text{objects}} \text{polygon count for object}$$

and m is the number of candidate polygons contained by the octree leaf.

However, decomposing a scene into an octree is an expensive operation and has to be judiciously controlled. It involves finding the 'minmax' coordinates of each polygon (the coordinates of its bounding box) and using these as an entity in the decomposition. Two factors that can be used to control the decomposition are:

(1) The minimum number of candidate polygons per node. The smaller this factor, the greater is the decomposition and fewer intersection tests are made by a ray that enters a voxel. The total number of intersection tests per voxel for the entire rendering is approximately given by:

number of rays entering the voxel × (0.5 × number of polygons in voxel)

assuming that on average a ray tests 50% of the candidate polygons before it finds an intersection.

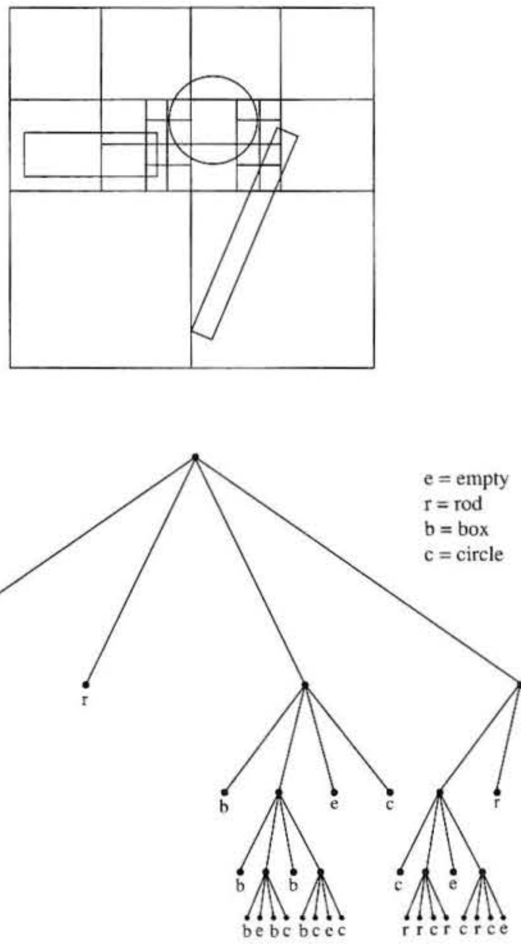


Figure 2.16
Quadtree representation of a two-dimensional scene down to the level of cells containing at most a single object. Terminal nodes for cells containing objects would be represented by a pointer to a data structure representation of the object.

(2) The maximum octree depth. The greater the depth the greater the decomposition and the fewer the candidate polygons at a leaf node. Also, because the size of a voxel decreases by a factor of 8 at every level, the fewer the rays that will enter the voxel for any given rendering.

In general the degree of decomposition should not be so great that the savings gained on intersection are wiped out by the higher costs of tracking a ray through decomposed space. Experience has shown that a default value of 8 for the above two factors gives good results in general for an object (or objects) distributed evenly throughout the space. Frequently scenes are rendered where this condition does not hold. Figure 2.17 shows an example where a few objects with high polygon count are distributed around a room whose volume is large compared to the space occupied by the objects. In this case octree subdivision will proceed to a high depth subdividing mostly empty space.

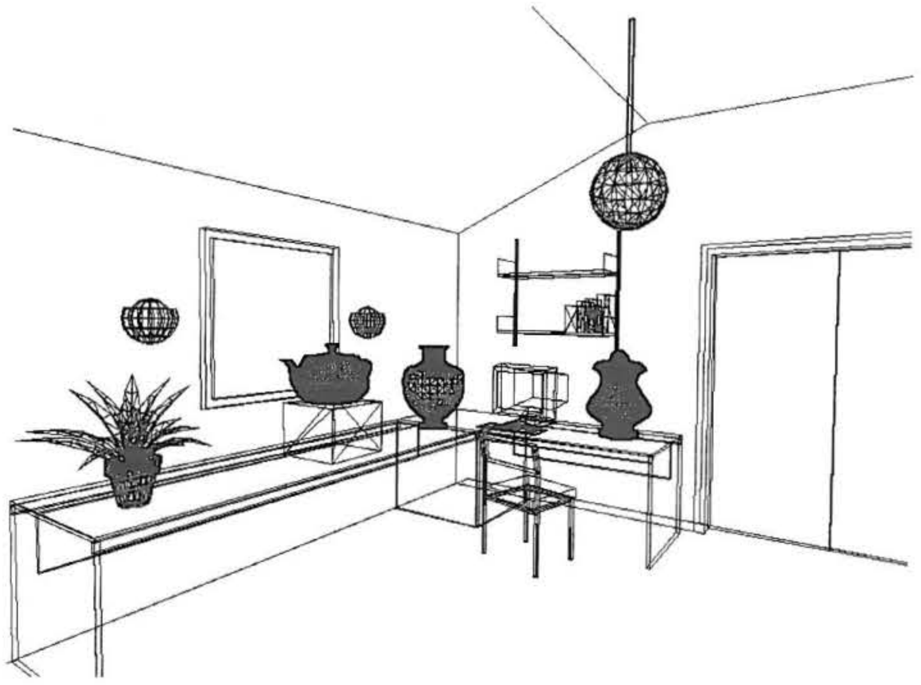


Figure 2.17
A scene consisting of a few objects of high polygon count. The objects are small compared with the volume of the room.

2.3.2 BSP trees

An alternative representation to an octree is a BSP or binary space partitioning tree. Each non-terminal node in the BSP tree represents a single partitioning plane that divides the space into two. A two-dimensional analogue illustrating the difference is shown in Figure 2.18. A BSP tree is not a direct object representation (although in certain circumstances it can be). Instead it is a way of partitioning space for a particular purpose – most commonly hidden surface removal. Because of this it is difficult and somewhat pointless to discuss BSP trees without dealing at the same time with HSR (see Chapter 6).

The properties of partitioning planes that can be exploited in computer graphics scenes are:

- Any object on one side of a plane cannot intercept any object on the other side.
- Given a view point in the scene space, objects on the same side as the viewer are nearer than any objects on the other side.

When a BSP tree is used to represent a subdivision of space into cubic cells, it shows no significant advantage over a direct data structure encoding of the octree. It is the same information encoded in a different way. However, nothing said above requires that the subdivision should be into cubic cells. In fact the

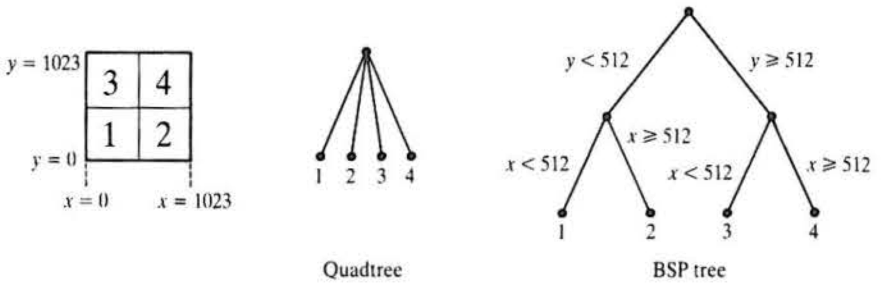


Figure 2.18
Quadtree and BSP tree representations of a one-level subdivision of a two-dimensional region.

idea of a BSP tree was originally introduced in Fuchs (1980) where the planes used to subdivide space could be at any orientation. We revisit BSP trees in the context of hidden surface removal (Chapter 6).

2.3.3 Creating voxel objects

One of the mainstream uses of voxel objects is in volume rendering in medical imagery. The source data in such applications consist of a set of parallel planes of intensity information collected from consecutive cross-sections from some part of a body, where a pixel in one such plane will represent, say, the X-ray absorption of that part of the body that the pixel physically corresponds to. The problem is how to convert such a stack of planar two-dimensional information into a three-dimensional rendered object. Converting the stack of planes to a set of voxels is the most direct way to solve this problem. Corresponding pixels in two consecutive planes are deemed to form the top and bottom face of a voxel and some operation is performed to arrive at a single voxel value from the two pixel values. The voxel representation is used as an intermediary between the raw collected data, which are two-dimensional, and the required three-dimensional visualization. The overall process from the collection of raw data, through the conversion to a voxel representation and the rendering of the voxel data is the subject of Chapter 13.

Contours collected by a laser ranger can be converted into a voxel representation instead of into a polygon mesh representation. However, this may result in a loss of accuracy compared with using a skinning algorithm.

2.4 Representing objects with implicit functions

As we have already pointed out, representing a whole object by a single implicit formula is restricted to certain objects such as spheres. Nevertheless such a representation does find mainstream use in representing 'algorithmic' objects known as bounding volumes. These are used in many different contexts in computer graphics as a complexity limiting device.

A representation developed from implicit formulae is the representation of objects by using the concept of implicitly defined objects as components. (We use the term component rather than primitive because the object is not simply a set of touching spheres but a surface derived from such a collection.)

Implicit functions are surfaces formed by the effect of primitives that exert a field of influence over a local neighbourhood. For example, consider a pair of point heat sources shown in Figure 2.19. We could define the temperature in their vicinities as a field function where, for each in isolation, we have isothermal contours as spherical shells centred on each source. Bringing the two sources within influence of each other defines a combined global scalar field, the field of each source combining with that of the other to form a composite set of isothermal contours as shown. Such a scalar field, due to the combined effect of a number of primitives is used to define a modelling surface in computer graphics. Usually we consider an isosurface in the field to be the boundary of a volume which is the object that we desire to model. Thus we have the following elements in any implicit function modelling system:

- A generator or primitive for which a distance function $d(\boldsymbol{P})$ can be defined for all points $\boldsymbol{P}$ in the locality of the generator.
- A 'potential' function $f(d(\boldsymbol{P}))$ which returns a scalar value for a point $\boldsymbol{P}$ distance $d(\boldsymbol{P})$ from the generator. Associated with the generator can be an area of influence outside of which the generator has no influence. For a point generator this is usually a sphere. An example of a potential function is:

  $$f(\boldsymbol{P}) = (1 - \frac{d^2}{R^2})^2 \qquad d \leq R$$

  where d is the distance of the point to the generator and R is its radius of influence.
- A scalar field $F(\boldsymbol{P})$ which determines the combined effect of the individual potential functions of the generators. This implies the existence of a blending method which in the simplest case is addition – we evaluate a scalar field by evaluating the individual contributions of each generator at a point $\boldsymbol{P}$ and adding their effects together.

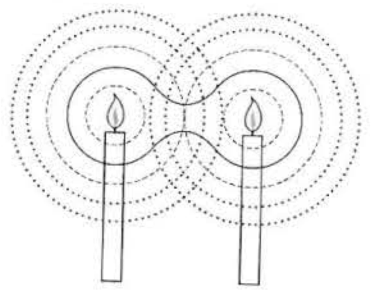


Figure 2.19
An isosurface of equal temperature around two heat sources (solid line).

- An isosurface of the scalar field which is used to represent the physical surface of the object that we are modelling.

An example (Figure 2.20 Colour Plate) illustrates the point. The Salvador Dali imitation on the left is an isosurface formed by point generators disposed in space as shown on the right. The radius of each sphere is proportional to the radius of influence of each generator. The dark spheres represent negative generators which are used to 'carve' concavities in the model. (Although we can form concavities by using only positive generators, it is more convenient to use negative ones as we require far fewer spheres.) The example illustrates the potential of the method for modelling organic shapes.

Deformable object animation can be implemented by displaying or choreographing the points that generate the object. The problem with using implicit functions in animation is that there is not a good intuitive link between moving groups of generators and the deformation that ensues because of this. Of course, this general problem is suffered by all modelling techniques where the geometry definition and the deformation method are one and the same thing.

In addition to this general problem, unwanted blending and unwanted separation can occur when the generators are moved with respect to each other and the same blending method retained.

A significant advantage of implicit functions in an animation context is the ease of collision detection that results from an easy inside–outside function. Irrespective of the complexity of the modelled surface a single scalar value defines the isosurface and a point $\boldsymbol{P}$ is inside the object volume or outside it depending on whether $F(\boldsymbol{P})$ is less than or greater than this value.

2.5 Scene management and object representation

As the demand for high quality real time computer graphics continues to grow, from applications like computer games and virtual reality, the issue of efficient scene management has become increasingly important. This means that representational forms have to be extended to collections of objects; in other words the scene has to be considered as an object itself. This has generally meant using hierarchical or tree structures, such as BSP trees to represent the scene down to object and sub-object level. As rendering has increasingly migrated into real time applications, efficiency in culling and hidden surface removal has become as important as efficient rendering for complex scenes. With the advent of 3D graphics boards for the PC we are seeing a trend develop where the basic rendering of individual objects is handled by hardware and the evaluation of which objects are potentially visible is computed by software. (We will look into culling and hidden surface removal in Chapters 5 and 6). An equally important efficiency measure for objects in complex scenes has come to be known as Level of Detail, or LOD, and it is this topic that we will now examine.

2.5.1 Polygon mesh optimization

As we have discussed, polygon mesh models are well established as the *de facto* standard representational form in computer graphics but they suffer from significant disadvantages, notably that the level of detail, or number of polygons, required to synthesize the object for a high quality rendition of a complex object is very large. If the object is to be rendered on screen at different viewing distances the pipeline has to process thousands of polygons that project onto a few pixels on the screen. As the projected polygon size decreases, the polygon overheads become significant and in real time applications this situation is intolerable. High polygon counts per object occur either because of object complexity or because of the nature of the modelling system. Laser scanners and the output from programs like the marching cubes algorithm (which converts voxels into polygons) are notorious for producing very large polygon counts. Using such facilities almost always results in a model that, when rendered, is indistinguishable from a version rendered from a model with far fewer faces.

As early as 1976, one of the pioneers of 3D computer graphics, James H. Clark, wrote:

It makes no sense to use 500 polygons in describing an object if it covers only 20 raster units of the display . . . For example, when we view the human body from a very large distance, we might need to present only specks for the eyes, or perhaps just a block for the head, totally eliminating the eyes from consideration . . . these issues have not been addressed in a unified way.

Did Clark realize that not many years after he had written these words that 500 000 polygon objects would become fairly commonplace and that complex scenes might contain millions of polygons?

Existing systems tend to address this problem in a somewhat ad hoc manner. For example, many cheap virtual reality systems adopt a two- or three-level representation switching in surface detail, such as the numbers on the buttons of a telephone as the viewer moves closer to it. This produces an annoying visual disturbance as the detail blinks on and off. More considered approaches are now being proposed and lately there has been a substantial increase in the number of papers published in this area.

Thus mesh optimization seems necessary and the problem cannot be dismissed by relying on increased polygon throughput of the workstations of the future. The position we are in at the moment is that mainstream virtual reality platforms produce a visually inadequate result even from fairly simple scenes. We have to look forward not only to dealing with the defects in the image synthesis of such scenes, but also to being able to handle scenes of real world complexity implying many millions of polygons. The much vaunted 'immersive' applications of virtual reality will never become acceptable unless we can cope with scenes of such complexity. Current hardware is very far away from being able to deal with a complex scene in real time to the level of quality attainable for single object scenes.

An obvious solution to the problem is to generate a polygon mesh at the final level of detail and then use this representation to spawn a set of coarser descriptions. As the scene is rendered an appropriate level of detail is selected. Certain algorithms have emerged from time to time in computer graphics that use this principle. An example of a method that facilitates a polygon mesh at any level of detail is bi-cubic parametric patches (see Section 4.2.2). Here we take a patch description and turn it into a polygon description. At the same time we can easily control the number of polygons that are generated for each patch and relate this to local surface curvature. This is exactly what is done in patch rendering where a geometric criterion is used to control the extent of the subdivision and produce an image free of geometric aliasing (visible polygon edges in silhouette). The price we pay for this approach is the expense and difficulty of getting the patch description in the first place. But in any case we could build the original patch representation and construct a pyramid of polygon mesh representations off-line.

The idea of storing a 'detail pyramid' and accessing an appropriate level is established in many application areas. Consider the case of mip-mapping, for example (see Chapter 8). Here texture maps are stored in a detail hierarchy and a fine detail map selected when the projection of the map on the screen is large. In the event that the map projects onto just one pixel, then a single pixel texture map – the average of the most detailed map – is selected. Also, in this method the problem of avoiding a jump when going from one level to another is carefully addressed and an approximation to a continuous level of detail is obtained by interpolation between two maps.

The diversity of current approaches underlines the relative newness of the field. A direct and simple approach for triangular meshes derived from voxel sets was reported by Schroeder *et al.* in 1992. Here the algorithm considers each vertex on a surface. By looking at the triangles that contribute to, or share, the vertex, a number of criteria can be enumerated and used to determine whether these triangles can be merged into a single one exclusive of the vertex under consideration. For example, we can invoke the 'reduce the number of triangles where the surface curvature is low' argument by measuring the variance in the surface normals of the triangles that share the vertex. Alternatively we could consider the distance from the vertex to an (average) plane through all the other vertices of the sharing triangles (Figure 2.21). This is a local approach that considers vertices in the geometry of their immediate surroundings.

A more recent approach is the work of Hoppe (1996) which we will now describe. Hoppe gives an excellent categorization of the problems and advantages of mesh optimization, listing these as follows:

- Mesh simplification – reducing the polygons to a level that is adequate for the quality required. (This, of course, depends on the maximum projection size of the object on the screen.)
- Level of detail approximation – a level is used that is appropriate to the viewing distance. In this respect, Hoppe adds: 'Since instantaneous switching between LOD meshes may lead to perceptible "popping", one would like to construct smooth visual transitions, *geomorphs*, between meshes at different resolutions'.
- Progressive transmission – this is a three-dimensional equivalent of the common progressive transmission modes used to transmit two-dimensional imagery over the Internet. Succesive LOD approximations can be transmitted and rendered at the reciever.
- Mesh compression – analagous to two-dimensional image pyramids. We can consider not only reducing the number of polygons but also minimizing the space that any LOD approximation occupies. As in two-dimensional imagery, this is important because an LOD hierarchy occupies much more memory than a single model stored at its highest level of detail.
- Selective refinement – an LOD representation may be used in a context-dependent manner. Hoppe gives the example of a user flying over a terrain where the terrain mesh need only be fully detailed near the viewer.

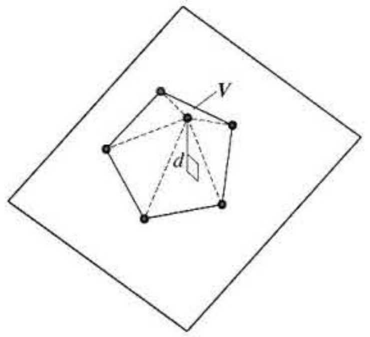


Figure 2.21
A simple vertex deletion criterion. Delete ***V***? Measure *d*, the distance from ***V*** to the (average) plane through the triangles that share ***V***.

Addressing mesh compression, Hoppe takes a 'pyramidal' approach and stores the coarsest level of detail approximation together, for each higher level, with the information required to ascend from a lower to a higher level of detail. To make the transition from a lower to a higher level the reverse of the transformation that constructed the hierarchy from the highest to the lowest level is stored and used. This is in the form of a vertex split – an operation that adds an additional vertex to the lower mesh to obtain the next mesh up the detail hierarchy. Although Hoppe originally considered three mesh transformations – an edge collapse, an edge split and an edge swap – he found that an edge collapse is sufficient for simplifying meshes.

The overall scheme is represented in Figure 2.22(a) which shows a detail pyramid which would be constructed off-line by a series of edge collapse transformations that take the original mesh M_n and generate through repeated edge collapse transformations the final or coarsest mesh M_0. The entire pyramid can then be stored as M_0 together with the information required to generate, from M_0 to any finer level M_i in the hierarchy – mesh compression. This inter-level transformation is the reverse of the edge collapse and is the information required

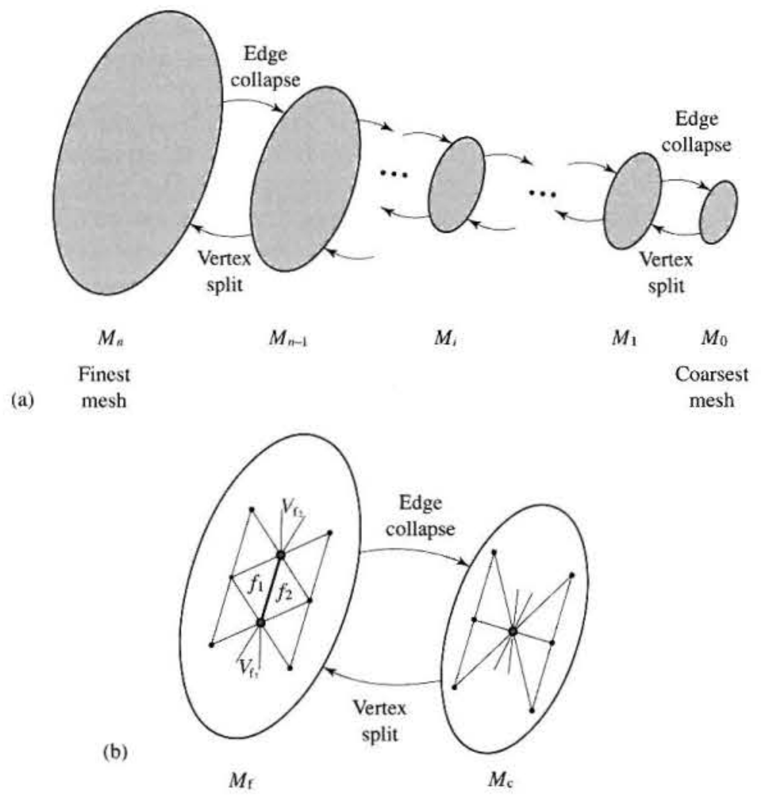


Figure 2.22
Hoppe's (1996) progressive mesh scheme based on edge collapse transformations.

for a vertex split. Hoppe quotes an example of an object with 13 546 faces which was simplified to an M_0 of 150 faces using 6698 edge collapse transformations. The original data are then stored as M_0 together with the 6698 vertex split records. The vertex split records themselves exhibit redundancy and can be compressed using classical data compression techniques.

Figure 2.22(b) shows a single edge collapse between two consecutive levels. The notation is as follows: V_{f1} and V_{f2} are the two vertices in the finer mesh that are collapsed into one vertex V_c in the coarser mesh, where

$$V_c \in \left\{ V_{f1},\ V_{f2},\ \frac{V_{f1} + V_{f2}}{2} \right\}$$

From the diagram it can be seen that this operation implies the collapse of the two faces f_1 and f_2 into new edges.

Hoppe defines a continuum between any two levels of detail by using a blending parameter α. If we define:

$$d = \frac{V_{f1} + V_{f2}}{2}$$

then we can generate a continuum of geomorphs between the two levels by having the edge shrink under control of the blending parameter as:

$$V_{f1} := V_{f1} + \alpha d \quad \text{and} \quad V_{f2} := V_{f2} - \alpha d$$

Texture coordinates can be interpolated in the same way as can scalar attributes associated with a vertex such as colour.

The remaining question is: how are the edges selected for collapse in the reduction from M_i to M_{i-1}? This can be done either by using a simple heuristic approach or by a more rigorous method that measures the difference between a particular approximation and a sample of the original mesh. A simple metric that can be used to order the edges for collapse is:

$$\frac{|V_{f1} - V_{f2}|}{|\mathbf{N}_{f1} \cdot \mathbf{N}_{f2}|}$$

that is, the length of the edge divided by the dot product of the vertex normals. On its own this metric will work quite well, but if it is continually applied the mesh will suddenly begin to 'collapse' and a more considered approach to edge selection is mandatory. Figure 2.23 is an example that uses this technique.

Hoppe casts this as an energy function minimzation problem. A mesh M is optimized with respect to a set of points X which are the vertices of the mesh M_n together (optionally) with points randomly sampled from its faces. (Although this is a lengthy process it is, of course, executed once only as an off-line preprocess.) The energy function to be minimized is:

$$E(M) = E_{\text{dist}}(M) + E_{\text{spring}}(M)$$

where

$$E_{\text{dist}} = \sum_i d^2(x_i, M)$$

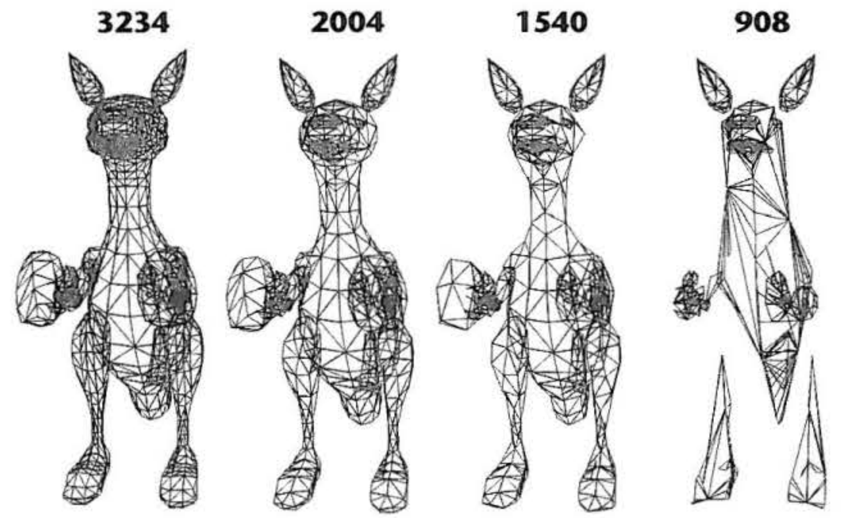


Figure 2.23
The result of applying the simple edge elimination criterion described in the text – the model eventually breaks up.

is the sum of the squared distances from the points X to the mesh – when a vertex is removed this term will tend to increase.

$$E_{\text{spring}}(M) = \sum \kappa \| v_j - v_k \|^2$$

is a spring energy term that assists the optimization. It is equivalent to placing on each edge a spring of rest length zero and spring constant κ.

Hoppe orders the optimization by placing all (legal) edge collapse transformations into a priority queue, where the priority of each transformation is its estimated energy cost ΔE. In each iteration, the transformation at the front of the queue (lowest ΔE) is performed and the priorities of the edges in the neighbourhood of this transformation are recomputed. An edge collapse transformation is only legal if it does not change the topology of the mesh. For example, if V_{f1} and V_{f2} are boundary vertices, the edge $\{V_{f1}, V_{f2}\}$ must be a boundary edge – it cannot be an internal edge connecting two boundary points.

2.6 Summary

Object representations have evolved under a variety of influences – ease of rendering, ease of shape editing, suitability for animation, dependence on the attributes of raw data and so on. There is no general solution that is satisfactory for all practical applications and the most popular solution that has served us for so many years – the polygon mesh – has significant disadvantages as soon as we leave the domain of static objects rendered off-line. We complete this chapter by listing the defining attributes of any representation. These allow a (very) general comparison between the methods. (For completeness we have included comments on bi-cubic patches which are dealt with in the next chapter.)

- **Creation of object/representation** A factor that is obviously context dependent. We have the methods which can create representations automatically from physical data (polygon mesh from range data via a skinning algorithm, bi-cubic parametric patches via interpolation of surface data). Other methods map input data directly into a voxel representation. Some methods are suitable for interactive creation (CSG and bi-cubic parametric patches) and some can be created by interacting with a 'mathematically' based interactive facility such as sweeping a cross-section along a spine (polygon mesh and bi-cubic parametric patches).
- **Nature of the primitive elements** The common forms are either methods that represent surfaces – boundary representations (polygon mesh and bi-cubic parametric patches) or volumes (voxels and CSG).
- **Accuracy** Representations are either exact or approximate. Polygon meshes are approximate representations but their accuracy can be increased to any degree at the expense of an expansion in the data. Increasing the accuracy of a polygon mesh representation in an intelligent way is difficult. The easy 'brute force' approach – throwing more polygons at the shape – may result in areas being 'over represented'. Bi-cubic patches can either be exact or approximate depending on the application. Surface interpolation will result in an approximation but designing a car door panel using a single patch results in an exact representation. CSG representations are exact but we need to make two qualifications. They can only describe that subset of shapes that is possible by combining the set of supplied primitives. The representation is abstract in that it is just a formula for the composite object – the geometry has to be derived from the formula to enable a visualization of the object.

- **Accuracy vs data volume** There is always a trade-off between accuracy and data volume – at least as far as the rendering penalty is concerned. To increase the accuracy of a boundary representation or a volume representation we have to increase the number of low-level elements. Although the implicit equation of a sphere is 100% accurate and compact, it has to be converted for rendering using some kind of geometric sampling procedure which generates low-level elements.
- **Data volume vs complexity** There is usually a trade-off also between data volume and the complexity of the representation which has practical ramifications in the algorithms that operate with the representation. This is best exemplified by comparing polygon meshes with their counterpart using bi-cubic parametric patches.
- **Ease of editing/animation** This can mean retrospective editing of an existing model or shape deformation techniques in an animation environment. The best method for editing the shape of static objects is, of course, the CSG representation – it was designed for this. Editing bi-cubic parametric patches is easy or difficult depending on the complexity of the shape and the desired freedom of the editing operations. In this respect editing a single patch is easy, editing a net of patches is difficult. None of the representation methods that we have described is suitable for shape-changing in animated sequences, although bi-cubic parametric patches and implicit functions have been tried. It seems that the needs of accuracy and ease of animating shape change are opposites. Methods that allow a high degree of accuracy are difficult to animate, because they consist of a structure with maybe thousands of low-level primitives as leaves. For example, the common way to control a net of bi-cubic parametric patches representing, say, the face of a character is to organize it into a hierarchy allowing local changes to be made (by descending the hierarchy and operating on a few or even a single patch) and making more global changes by operating at a high level in the structure. This has not resulted in a generally accepted animation technique simply because it does not produce good results (in the case of facial animation anyway). It seems shape-change animation needs a paradigm that is independent of the object model and the most successful techniques involve embedding the object model in *another* structure which is then subject to shape-change animation. Thus we control facial animation by attaching a geometric structure to a muscle control model or immerse a geometric model in the 'field' of an elastic solid and animate the elastic solid. In other words for the representations that we currently use, animation of shape does not seem to be possible by operating directly on the geometry of the object.

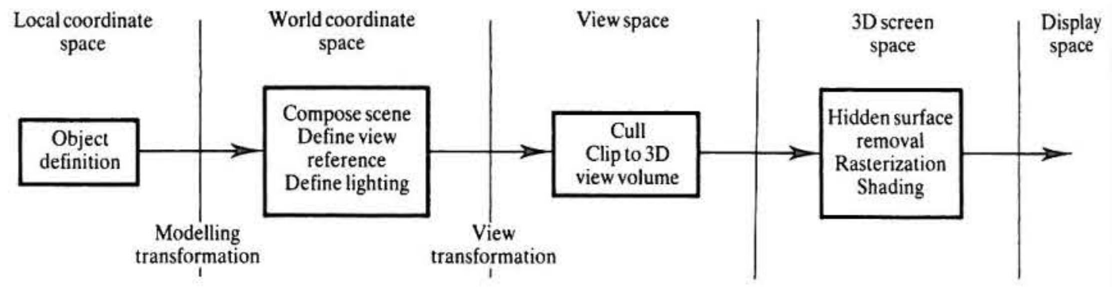
5 The graphics pipeline (1): geometric operations

Introduction

The purpose of a graphics pipeline is to take a description of a scene in three-dimensional space and map it into a two-dimensional projection on the view surface – the monitor screen. Although various three-dimensional display devices exist, most computer graphics view surfaces are two-dimensional, the most common being the TV-type monitor. In most VR applications a pair of projections is produced and displayed on small monitors encased in a helmet – a head-mounted display, or HMD. The only difference in this case is that we now have a pair of two-dimensional projections instead of one – the operations remain the same.

In what follows we will consider the case of polygon mesh models. We can loosely classify the various processes involved in the graphics pipeline by putting them into one of two categories – geometric (this chapter) and algorithmic (Chapter 6). Geometric processes involve operations on the vertices of polygons – transforming the vertices from one coordinate space to another or discarding

Figure 5.1
A three-dimensional rendering pipeline.



polygons that cannot be seen from the view point, for example. Rendering processes involve operations like shading and texture mapping and are considerably more costly than the geometric operations most of which involve matrix multiplication.

A diagram representing the consecutive process in a graphics pipeline is shown in Figure 5.1. From this it can be seen that the overall process is a progression through various three-dimensional spaces – we transform the object representation from space to space. In the final space, which we have called screen space, the rendering operations take place. This space is also three-dimensional for reasons that we will shortly discover.

5.1 Coordinate spaces in the graphics pipeline

5.1.1 Local or modelling coordinate systems

For ease of modelling it makes sense to store the vertices of a polygon mesh object with respect to some point located in or near the object. For example, we would almost certainly want to locate the origin of a cube at one of the cube vertices, or we would want to make the axis of symmetry of an object generated as a solid of revolution, coincident with the *z* axis. As well as storing the polygon vertices in a coordinate system that is local to the object, we would also store the polygon normal and the vertex normals. When local transformations are applied to the vertices of an object, the corresponding transformations are applied to the associated normals.

5.1.2 World coordinate systems

Once an object has been modelled the next stage is to place it in the scene that we wish to render. All objects that together constitute a scene have their separate local coordinate systems. The global coordinate system of the scene is known as the 'world coordinate system'. All objects have to be transformed into this common space in order that their relative spatial relationships may be defined. The act of placing an object in a scene defines the transformation required to take the object from local space to world space. If the object is being animated, then the animation system provides a time-varying transformation that takes the object into world space on a frame by frame basis.

The scene is lit in world space. Light sources are specified, and if the shaders within the renderer function are in world space then this is the final transformation that the normals of the object have to undergo. The surface attributes of an object – texture, colour and so on – are specified and tuned in this space.

5.1.3 Camera or eye or view coordinate system

The eye, camera or view coordinate system is a space that is used to establish viewing parameters (view point, viewing direction) and a view volume. (A virtual camera is often used as the analogy in viewing systems, but if such an allusion is made we must be careful to distinguish between external camera parameters – its position and the direction it is pointing in – and internal camera parameters or those that affect the nature and size of the image on the film plane. Most rendering systems imitate a camera which in practice would be a perfect pinhole (or lensless) device with a film plane that can be positioned at any distance with respect to the pinhole. However, there are other facilities in computer graphics that cannot be imitated by a camera and because of this the analogy is of limited utility.)

We will now deal with a basic view coordinate system and the transformation from world space to view coordinate space. The reasons that this space exists, after all we could go directly from world space to screen space, is that certain operations (and specifications) are most conveniently carried out in view space. Standard viewing systems like that defined in the PHIGS graphics standard are more complicated in the sense that they allow the user to specify more facilities and we will deal with these in Section 5.3.

We define a viewing system as being the combination of a view coordinate system together with the specification of certain facilities such as a view volume. The simplest or minimum system would consist of the following:

- A view point which establishes the viewer's position in world space; this can either be the origin of the view coordinate system or the centre of projection together with a view direction ***N***.
- A view coordinate system defined with respect to the view point.
- A view plane onto which the two-dimensional image of the scene is projected.
- A view frustum or volume which defines the field of view.

These entities are shown in Figure 5.2. The view coordinate system, ***UVN***, has ***N*** coincident with the viewing direction and ***V*** and ***U*** lying in a plane parallel to the view plane. We can consider the origin of the system to be the view point ***C***. The view plane containing ***U*** and ***V*** is of infinite extent and we specify a view volume or frustum which defines a window in the view plane. It is the contents of this window – the projection of that part of the scene that is contained within the view volume – that finally appears on the screen.

Thus, using the virtual camera analogue we have a camera that can be positioned anywhere in world coordinate space, pointed in any direction and rotated about the viewing direction ***N***.

To transform points in world coordinate space we invoke a change of coordinate system transformation and this splits into two components: a translational one and a rotational one (see Chapter 1). Thus:

$$\begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix} = \boldsymbol{T}_{\text{view}} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}$$

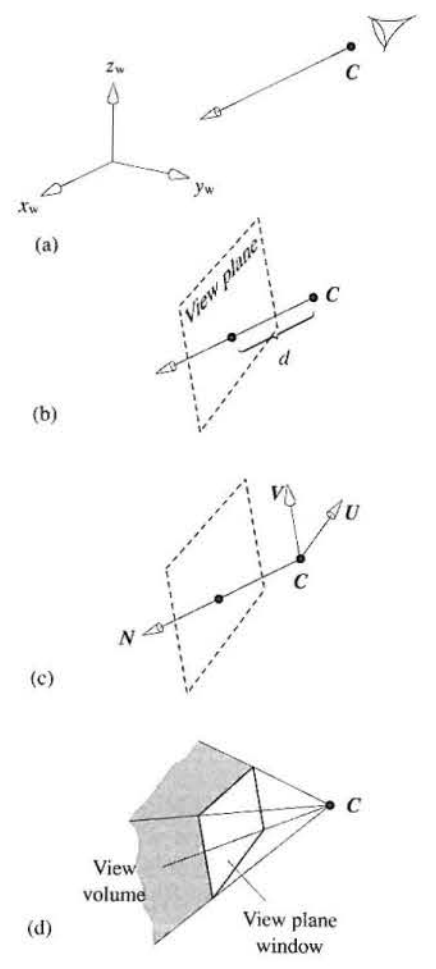


Figure 5.2 The minimum entities required in a practical viewing system. (a) View point ***C*** and viewing direction ***N***. (b) A view plane normal to the viewing direction ***N*** positioned *d* units from ***C***. (c) A view coordinate system with the origin ***C*** and ***UV*** axes embedded in plane parallel to the view plane. (d) A view volume defined by the frustum formed by ***C*** and the view plane window.

where:

$$\boldsymbol{T}_{\text{view}} = \boldsymbol{RT}$$

and:

$$\boldsymbol{T} = \begin{bmatrix} 1 & 0 & 0 & -C_x \\ 0 & 1 & 0 & -C_y \\ 0 & 0 & 1 & -C_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \boldsymbol{R} = \begin{bmatrix} U_x & U_y & U_z & 0 \\ V_x & V_y & V_z & 0 \\ N_x & N_y & N_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The only problem now is specifying a user interface for the system and mapping whatever parameters are used by the interface into ***U***, ***V*** and ***N***. A user needs to specify ***C***, ***N*** and ***V***. ***C*** is easy enough. ***N***, the viewing direction or view plane normal, can be entered say, using two angles in a spherical coordinate system – this seems reasonably intuitive:

θ the azimuth angle
ϕ the colatitude or elevation angle

where:

$N_x = \sin\phi \cos\theta$
$N_y = \sin\phi \sin\theta$
$N_z = \cos\phi$

V is more problematic. For example, a user may require 'up' to be the same sense as 'up' in the world coordinate system. However, this cannot be achieved by setting:

$\boldsymbol{V} = (0, 0, 1)$

because ***V*** must be perpendicular to ***N***. A sensible strategy is to allow a user to specify an approximate orientation for ***V***, say ***V***' and have the system calculate ***V***. Figure 5.3 demonstrates this. ***V***' is the user-specified up vector. This is projected onto the view plane:

$\boldsymbol{V} = \boldsymbol{V}' - (\boldsymbol{V}'.\boldsymbol{N})\boldsymbol{N}$

and normalized. ***U*** can be specified or not depending on the user's requirements. If ***U*** is unspecified, it is obtained from:

$\boldsymbol{U} = \boldsymbol{N} \times \boldsymbol{V}$

This results in a left-hand coordinate system, which although somewhat inconsistent, conforms with our intuition of a practical viewing system, which has increasing distances from the view point as increasing values along the view direction axis. Having established the viewing transformation using ***UVN*** notation, we will in subsequent sections use (x_v, y_v, z_v) to specify points in the view coordinate system.

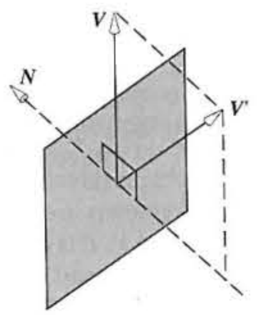


Figure 5.3
The up vector ***V*** can be calculated from an 'indication' given by ***V***'.

5.2 Operations carried out in view space

5.2.1 Culling or back-face elimination

Culling or back-face elimination is an operation that compares the orientation of complete polygons with the view point or centre of projection and removes those polygons that cannot be seen. If a scene only contains a single convex object, then culling generalizes to hidden surface removal. A general hidden surface removal algorithm is always required when one polygon partially obscures another (Figure 5.4). On average, half of the polygons in a polyhedron are back-facing and the advantage of this process is that a simple test removes these polygons from consideration by a more expensive hidden surface removal algorithm.

The test for visibility is straightforward and is best carried out in view space. We calculate the outward normal for a polygon (see Section 1.3.3) and examine the sign of the dot product of this and the vector from the centre of projection (Figure 5.5). Thus:

visibility:= $\boldsymbol{N}_p \cdot \boldsymbol{N} > 0$

where:

$\boldsymbol{N}_p$ is the polygon normal
N is the 'line of sight' vector

5.2.2 The view volume

In Figure 5.2 the view volume was introduced as a semi-infinite pyramid. In many applications this is further constrained to a general view volume which is defined by a view plane window, a near clip plane and a far clip plane, but to simplify matters we will dispense with the near clip plane, which is of limited practical utility, and reconsider a view volume defined only by a view plane window and a far clip plane (Figure 5.6). Note that the far clip plane is a cut-off plane normal to the viewing direction and any objects beyond this cannot be seen. In effect we have turned a semi-infinite pyramid into an infinite one. Far clip planes are extremely useful and can be used to cut down the number of polygons that need to be processed when rendering a very complex scene. For example, when flying through an environment in a three-dimensional computer

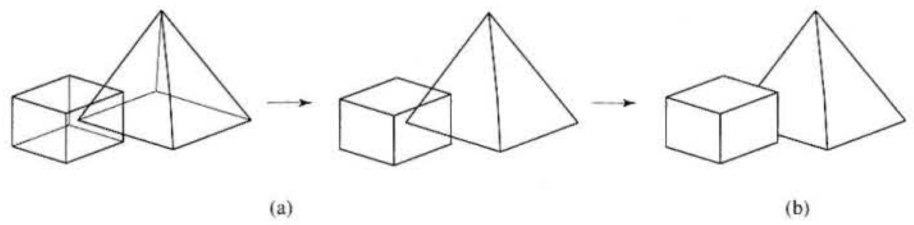


Figure 5.4
Culling and hidden space removal. (a) Culling removes complete polygons that cannot be seen.
(b) Hidden surface removal deals with the general problem: polygons will partially obscure others.

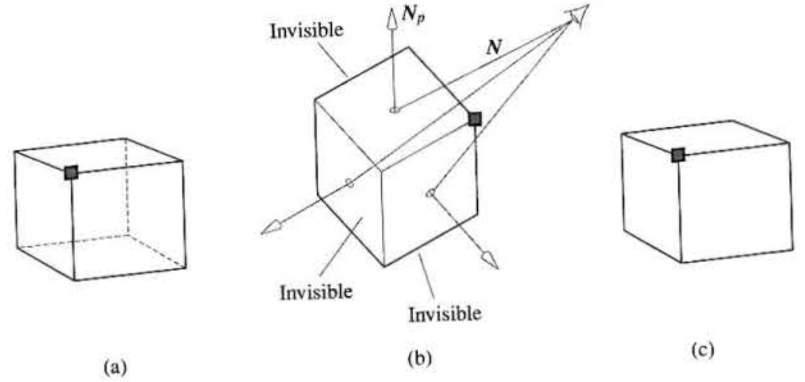


Figure 5.5
Culling or back-face elimination. (a) The desired view of the object (back faces shown as dotted lines). (b) A view of the geometry of the culling operation. (c) The culled object.

game we may specify a far clip plane and use depth modulated fog to diminish the disturbance as objects 'switch-on' when they suddenly intersect the far clip plane.

If we further simplify the geometry by specifying a square view plane window of dimension $2h$ arranged symmetrically about the viewing direction, then the four planes defining the sides of the view volume are given by:

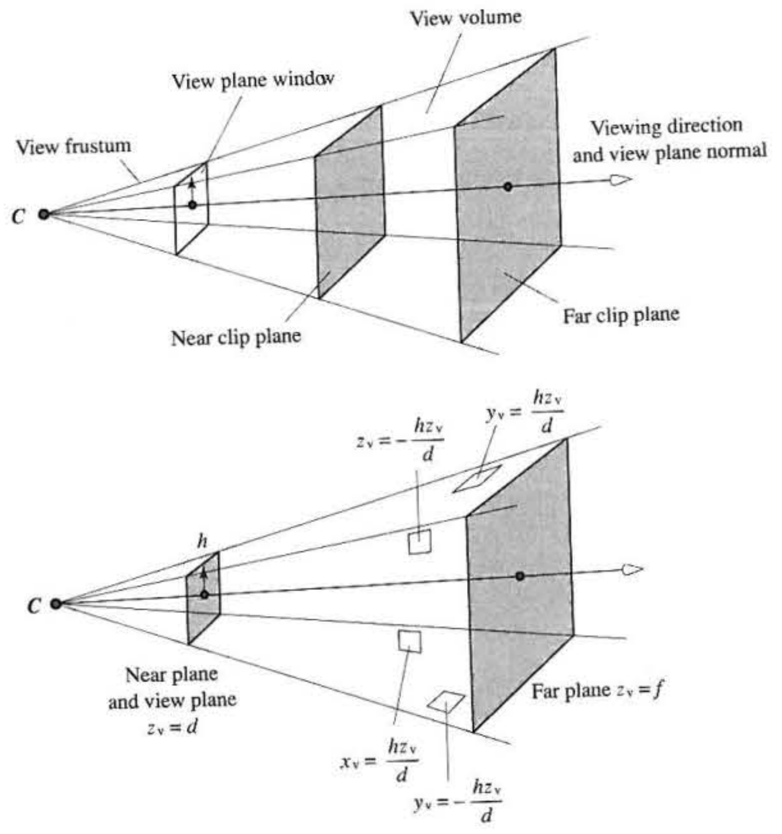


Figure 5.6
A practical view volume: the near clip plane is made coincident with the view plane.

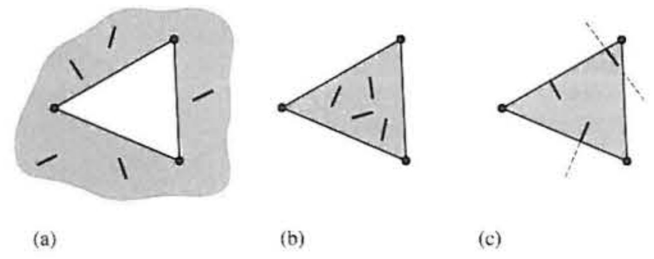


Figure 5.7
Clipping against a view volume – a routine polygon operation in the pipeline. (a) Polygons outside the view volume are discarded. (b) Polygons inside the view volume are retained. (c) Polygons intersecting a boundary are clipped.

$$x_v = \pm\frac{hz_v}{d}$$
$$y_v = \pm\frac{hz_v}{d}$$

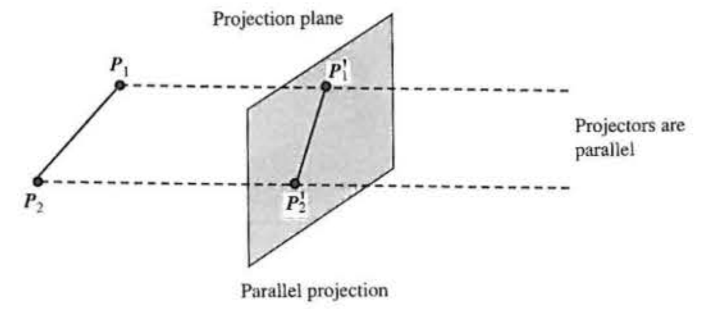
Clipping against the view volume (Figure 5.7) can now be carried out using polygon plane intersection calculations given in Section 1.4.3. This illustrates the principle of clipping, but the calculations involved are more efficiently carried out in three-dimensional screen space as we shall see.

5.2.3 Three-dimensional screen space

The final three-dimensional space in our pipeline we call three-dimensional screen space. In this space we carry out (practical) clipping against the view volume and the rendering processes that we will describe later. Three-dimensional screen space is used because it simplifies both clipping and hidden surface removal – the classic hidden surface removal algorithm being the Z-buffer algorithm which operates by comparing the depth values associated with different objects that project onto the same pixel. Also in this space there is a final transformation to two-dimensional view plane coordinates – sometimes called the perspective divide. (The terms 'screen' and 'view plane' mean slightly different things. Strictly speaking screen coordinates are derived from view plane coordinates by a device-dependent transformation.)

Because the viewing surface in computer graphics is deemed to be flat we consider the class of projections known as planar geometric projections. Two basic projections, perspective and parallel, are now described. These projections and the difference in their nature is illustrated in Figure 5.8.

A perspective projection is the more popular or common choice in computer graphics because it incorporates foreshortening. In a perspective projection relative dimensions are not preserved, and a distant line is displayed smaller than a nearer line of the same length (Figure 5.9). This effect enables human beings to perceive depth in a two-dimensional photograph or a stylization of three-dimensional reality. A perspective projection is characterized by a point known



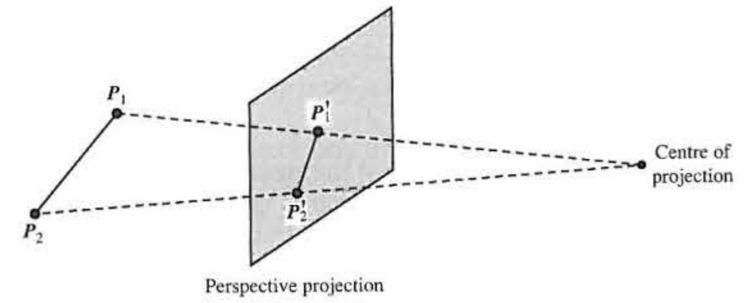


Figure 5.8
Two points projected onto a plane using parallel and perspective projections.

as the centre of projection and the projection of three-dimensional points onto the view plane is the intersection of the lines from each point to the centre of projection. These lines are called projectors.

Figure 5.10 shows how a perspective projection is derived. Point $\boldsymbol{P}$ (x_v, y_v, z_v) is a three-dimensional point in the view coordinate system. This point is to be projected onto a view plane normal to the z_v axis and positioned at distance d from the origin of this system. Point $\boldsymbol{P}'$ is the projection of this point in the view plane and has two-dimensional coordinates (x_s, y_s) in a view plane coordinate system with the origin at the intersection of the z_v axis and the view plane.

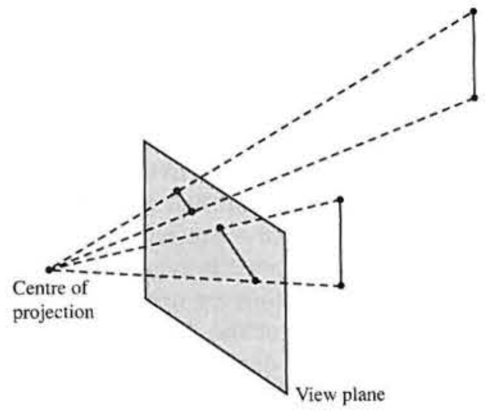
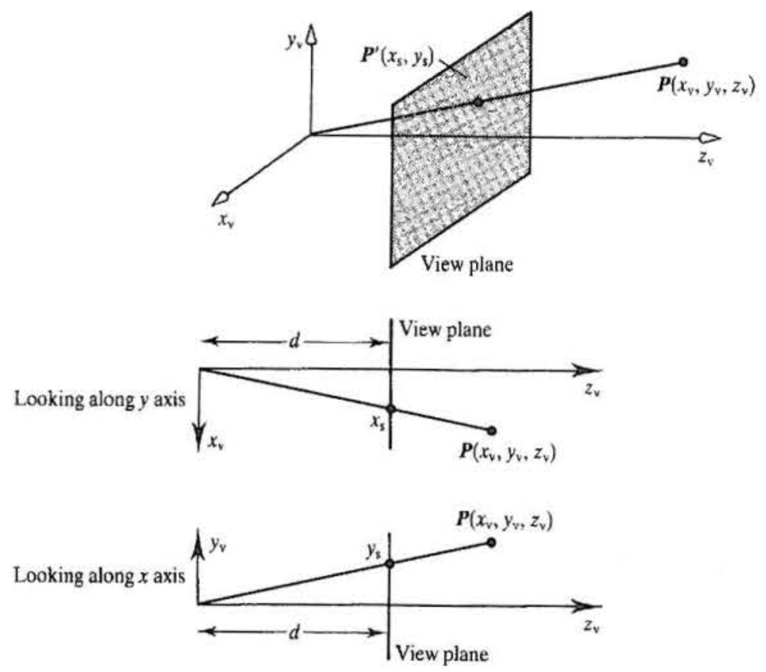


Figure 5.9
In a perspective projection a distant line is displayed smaller than a nearer line the same length.

Figure 5.10
Deriving a perspective transformation.



Similar triangles give:

$$\frac{x_s}{d} = \frac{x_v}{z_v} \qquad \frac{y_s}{d} = \frac{y_v}{z_v}$$

To express this non-linear transformation as a 4 × 4 matrix we can consider it in two parts – a linear part followed by a non-linear part. Using homogeneous co-ordinates we have:

$$X = x_v$$
$$Y = y_v$$
$$Z = z_v$$
$$w = z_v/d$$

We can now write:

$$\begin{bmatrix} X \\ Y \\ Z \\ w \end{bmatrix} = \boldsymbol{T}_{\text{pers}} \begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix}$$

where:

$$\boldsymbol{T}_{\text{pers}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

following this with the perspective divide, we have:

$$x_s = X/w$$
$$y_s = Y/w$$
$$z_s = Z/w$$

In a parallel projection, if the view plane is normal to the direction of projection then the projection is orthographic and we have:

$$x_s = x_v \quad y_s = y_v \quad z_v = 0$$

Expressed as a matrix:

$$\boldsymbol{T}_{\text{ort}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

5.2.4 View volume and depth

We now consider extending the above simple transformations to include the simplified view volume introduced in Figure 5.6. We discuss in more detail the transformation of the third component of screen space, namely z_s – ignored so far because the derivation of this transformation is somewhat subtle. Now, the bulk of the computation involved in rendering an image takes place in screen space. In screen space polygons are clipped against scan lines and pixels, and hidden surface calculations are performed on these clipped fragments. In order to perform hidden surface calculations (in the Z-buffer algorithm) depth information has to be generated on arbitrary points within the polygon. In practical terms this means, given a line and plane in screen space, being able to intersect the line with the plane, and to interpolate the depth of this intersection point, lying on the line, from the depth of the two end points. This is only a meaningful operation in screen space providing that in moving from eye space to screen space, lines transform into lines and planes transform into planes. It can be shown (Newman and Sproull 1973) that these conditions are satisfied provided the transformation of z takes the form:

$$z_s = A + B/z_v$$

where A and B are constants. These constants are determined from the following constraints:

(1) Choosing $B < 0$ so that as z_v increases then so does z_s. This preserves our intuitive Euclidean notion of depth. If one point is behind another, then it will have a larger z_v value, if $B < 0$ it will also have a larger z_s value.

(2) An important practical consideration concerning depth is the accuracy to which we store its value. To ensure this is as high as possible we normalize the range of z_s values so that the range $z_v \in [d, f]$ maps into the range $z_s \in [0, 1]$.

Considering the view volume in Figure 5.6, the full perspective transformation is given by:

$$x_s = d\frac{x_e}{hz_v}$$
$$y_s = d\frac{y_e}{hz_v}$$
$$z_s = \frac{f(1 - d/z_v)}{(f - d)}$$

where the additional constant, h, appearing in the transformation for x_s and y_s ensures that these values fall in the range [–1, 1] over the square screen. Adopting a similar manipulation to Section 5.2.3, we have:

$$X = \frac{d}{h}\, x_v$$
$$Y = \frac{d}{h}\, y_v$$
$$Z = \frac{fz_v}{f-d} - \frac{df}{f-d}$$
$$w = z_v$$

giving:

$$\begin{bmatrix} X \\ Y \\ Z \\ w \end{bmatrix} = \boldsymbol{T}_{\text{pers}} \begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix}$$

where:

$$\boldsymbol{T}_{\text{pers}} = \begin{bmatrix} d/h & 0 & 0 & 0 \\ 0 & d/h & 0 & 0 \\ 0 & 0 & f/(f-d) & -df/(f-d) \\ 0 & 0 & 1 & 0 \end{bmatrix} \qquad [5.1]$$

We can now express the overall transformation from world space to screen space as a single transformation obtained by concatenating the view and perspective transformation giving:

$$\begin{bmatrix} X \\ Y \\ Z \\ w \end{bmatrix} = \boldsymbol{T}_{\text{pers}}\boldsymbol{T}_{\text{view}} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}$$

It is instructive to consider the relationship between z_v and z_s a little more closely; although as we have seen by construction, they both provide a measure of the depth of a point, interpolating along a line in eye space is not the same as interpolating this line in screen space. Figure 5.11 illustrates this point. Equal

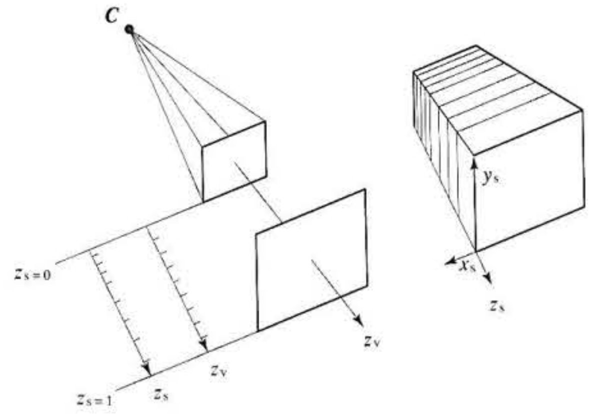


Figure 5.11
Illustrating the distortion in three-dimensional screen space due to the z_v to z_s transformation.

intervals in z_v are compared with the corresponding intervals in z_s. As z_v approaches the far clipping plane, z_s approaches 1 more rapidly. Thus, objects in screen space get pushed and distorted towards the back of the viewing frustum. This difference can lead to errors when interpolating quantities, other than position, in screen space.

In spite of this difficulty, by its very construction screen space is eminently suited to perform the hidden surface calculation. All rays passing through the view point are now parallel to the z_s axis because the centre of projection has been moved to negative infinity along the z_s axis. This can be seen by putting $z_v = 0$ into the above equation giving $z_s = -\infty$. Making those rays that hit the eye parallel, in screen space, means that hidden surface calculation need only be carried out on those points that have the same (x_s, y_s) coordinates. The test reduces to a simple comparison between z_s values to tell if a point is in front of another.

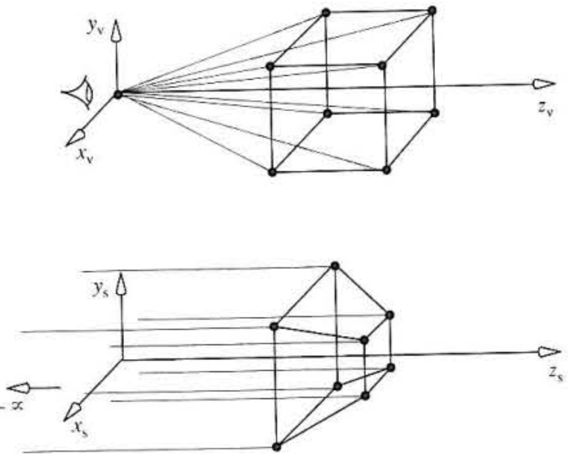


Figure 5.12
Transformation of box and light rays from eye space to screen space.

The transformation of a box with one side parallel to the image plane is shown in Figure 5.12. Here rays from the vertices of the box to the view point become parallel in three-dimensional screen space.

The overall precision required for the screen depth is a function of scene complexity. For most scenes 8 bits is insufficient and 16 bits usually suffices. The effects of insufficient precision is easily seen when, for example, a Z-buffer algorithm is used in conjunction with two intersecting objects. If the objects exhibit a curve where they intersect, this will produce aliasing artefacts of increasing severity as the precision of the screen depth is reduced.

Now return to the problem of clipping. It is easily seen from Figure 5.12 that in the homogeneous coordinate representation of screen space the sides of the view volume are parallel. This means that clipping calculations reduce to limit comparisons – we no longer have to substitute points into plane equations. The clipping operations must be performed on the homogeneous coordinates before the perspective divide, and translating the definition of the viewing frustum into homogeneous coordinates gives us the clipping limits:

$$-w \le x \le w$$
$$-w \le y \le w$$
$$0 \le z \le w$$

It is instructive to also consider the view space to eye space transformation by splitting Equation 5.1 into a product:

$$\boldsymbol{T}_{\text{pers}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & f/(f-d) & -df/(f-d) \\ 0 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} d/h & 0 & 0 & 0 \\ 0 & d/h & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \boldsymbol{T}_{\text{pers2}}\boldsymbol{T}_{\text{pers1}}$$

This enables a useful visualization of the process. The first matrix is a scaling (d/h) in x and y. This effectively converts the view volume from a truncated pyramid with sides sloping at an angle determined by h/d into a regular pyramid with sides sloping at 45° (Figure 5.13). For example, point:

$(0, h, d, 1)$ transforms to $(0, d, d, 1)$

and point:

$(0, -h, d, 1)$ transforms to $(0, -d, d, 1)$

The second transformation maps the regular pyramid into a box. The near plane maps into the (x, y) plane and the far plane is mapped into z=1. For example, point:

$(0, d, d, 1)$ transforms to $(0, d, 0, d)$

which is equivalent to $(0, 1, 0, 1)$.

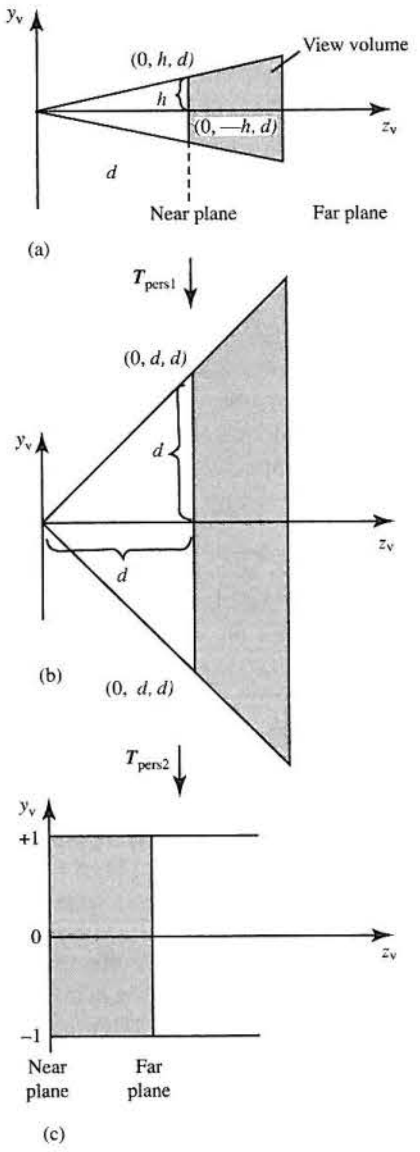


Figure 5.13 Transformation of the view volume into a canonical view volume (a box) using two matrix transformations.

5.3 Advanced viewing systems (PHIGS and GKS)

The viewing systems defined by the graphics standards PHIGS and GKS are far more general and more difficult to implement and understand than the system just described and so this section is very much optional reading. An unfortunate aspect of the standards viewing systems is that because they afford such generality they are hopelessly cumbersome and difficult to interface with. Even if a subset of parameters is used the default values for the unused parameters have

to be appreciated and understood. Perhaps this is inevitable. The function of a standard, in one sense, is not to reflect common usage but to define a complete set of facilities that a user may require in a general viewing system. That some of these facilities will hardly ever be used is unfortunate. We include the PHIGS viewing system as a topic for study because whatever viewing system you encounter, it will undoubtedly be a subset of this one.

5.3.1 Overview of the PHIGS viewing system

We start this section by overviewing the extensions that PHIGS offers over the minimal viewing system described in the previous section. These are:

(1) The notion of a view point or eye point that establishes the origin of the view coordinate system and the centre of projection is now discarded. The equivalent of view space in PHIGS is the view reference coordinate system (VRC) established by defining a view reference point (VRP). A centre of projection is established separately by defining a projection reference point (PRP).

(2) Feature (1) means that a line from the centre of projection to the centre of the view plane window need not be parallel to the view plane normal. The implication of this is that oblique projections are possible. This is equivalent, in the virtual camera analogy, to allowing the film plane to tilt with respect to the direction in which the camera is pointing. This effect is used in certain camera designs to correct for perspective distortion in such contexts as, for example, photographing a tall building from the ground.

(3) Near and far clipping planes are defined as well as a view plane. In the previous viewing system we made the back clipping plane coincident with the view plane.

(4) A view plane window is defined that can have any aspect ratio and can be positioned anywhere in the view plane. In the previous viewing system we defined a square window symmetrically disposed about the 'centre' of the view plane.

(5) Multiple view reference coordinate systems can be defined or many views of a scene can be set up.

Consider the notion of distance in viewing systems. We have previously used the idea of a view point distance to reflect the dominant intuitive notion that the further the view point is from the scene the smaller will be the projection of that scene on the view plane. The problem arises from the fact that in any real system, or a general computer graphics system, there is no such thing as a view point. We can have a centre of projection and a view plane, and in a camera or eye analogy this is fine. In a camera the view plane or film plane is contained in the camera. The scene projection is determined both by the distance of the camera from the subject and the focal length of the lens. However, in computer graphics we are free to move the view plane at will with respect to the centre of projection and the scene. There is no lens as such. How then do we categorize distance? Do we use the distance of the view plane from the world coordinate origin, the distance of the centre of projection from the world coordinate origin or the distance of the view plane from the centre of projection? The general systems such as PHIGS leaves the user to answer that question. It is perhaps this attribute of the PHIGS viewing system that makes it appear cumbersome.

PHIGS categorizes a viewing system into three stages (Figure 5.14). Establishing the position and orientation of the view plane is known as view orientation. This requires the user to supply:

(1) The view reference point (VRP) – a point in world coordinate space.

(2) The viewing direction or view plane normal (VPN) – a vector in world coordinate space.

(3) The view up vector (VUV) – vector in world coordinate space.

The second stage in the process is known as view mapping and determines how points are mapped onto the view plane. This requires:

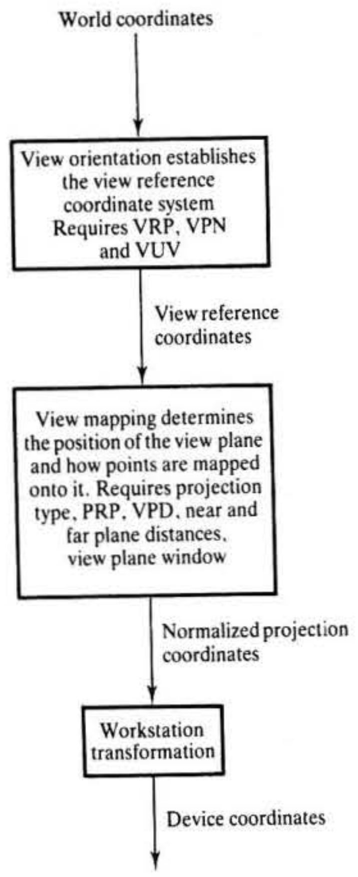

World coordinates
View orientation establishes the view reference coordinate system Requires VRP, VPN and VUV
View reference coordinates
View mapping determines the position of the view plane and how points are mapped onto it. Requires projection type, PRP, VPD, near and far plane distances, view plane window
Normalized projection coordinates
Workstation transformation
Device coordinates

Figure 5.14 Establishing a viewing system PHIGS.

(1) The projection type (parallel or perspective).

(2) The projection reference point (PRP) – a point in VRC space.

(3) A view plane distance (VPD) – a real in VRC space.

(4) A back plane and a front plane distance – reals in VRC space.

(5) View plane window limits – four limits in VRC space.

(6) Projection viewport limits – four limits in normalized projection space (NPC).

This information is used to map information in the VRC or VRCs into normalized projection coordinates (NPC). NPC space is a cube with coordinates in each direction restricted to the range 0 to 1. The rationale for this space is to allow different VRCs to be set up when more than one view of a scene is required (and mapped subsequently into different view ports on the screen). Each view has its own VRC associated with it, and different views are mapped into NPC space.

The final processing stage is the workstation transformation, that is, the normal device dependent transformation.

We now describe these aspects in detail.

5.3.2 The view orientation parameters

In PHIGS we establish a view space coordinate system whose origin is positioned anywhere in world coordinate space by the VRP. Together with the VPN and the VUV this defines a right-handed coordinate system with axes U, V and N. N is the viewing direction and UV defines a plane that is either coincident or parallel to the view plane. (The VUV has exactly the same function as V' in Section 5.1.3). The cross-product of VUV and VPN defines U and the cross-product of VPN and U defines V:

$$U = (\text{VUV}) \times (\text{VPN})$$
$$N = (\text{VPN}) \times U$$

An interface to establish the VPN can easily be set up using the suggestion given in Section 5.1.3. (Note that the VUV must not be parallel to the VPN.) The geometric relationship between the orientation and mapping parameters is shown in Figure 5.15. Thus the view orientation stage establishes the position and orientation of the VRC relative to the world coordinate origin and the view plane specification is defined relative to the VRC.

5.3.3 The view mapping parameters

The view mapping stage defines how the scene, and what part of it, is projected onto the view plane. In the view orientation process the parameters were defined in world coordinate space. In this stage the parameters are defined in view space.

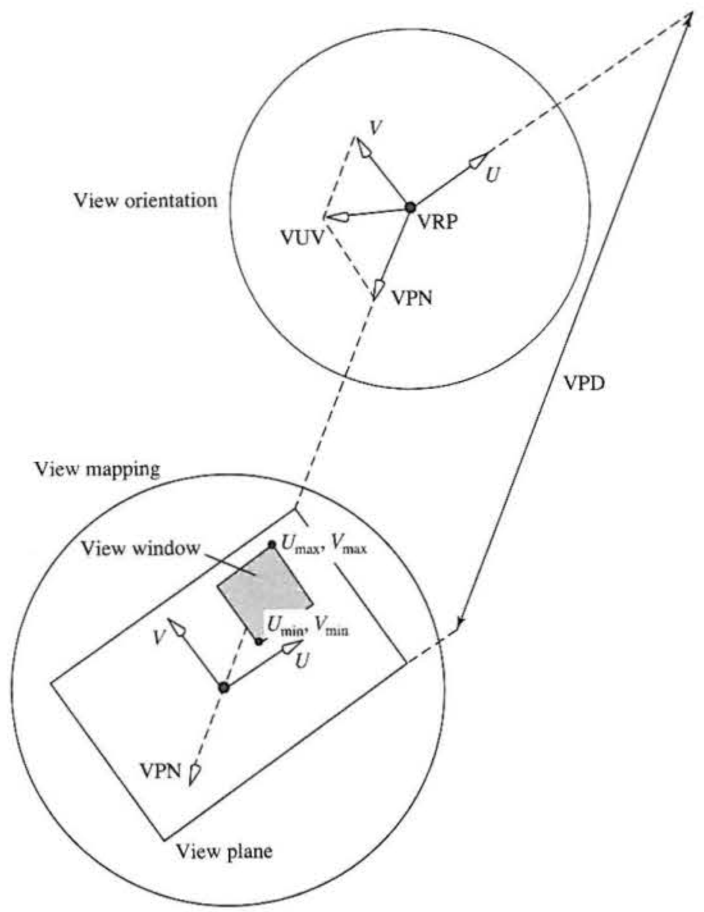


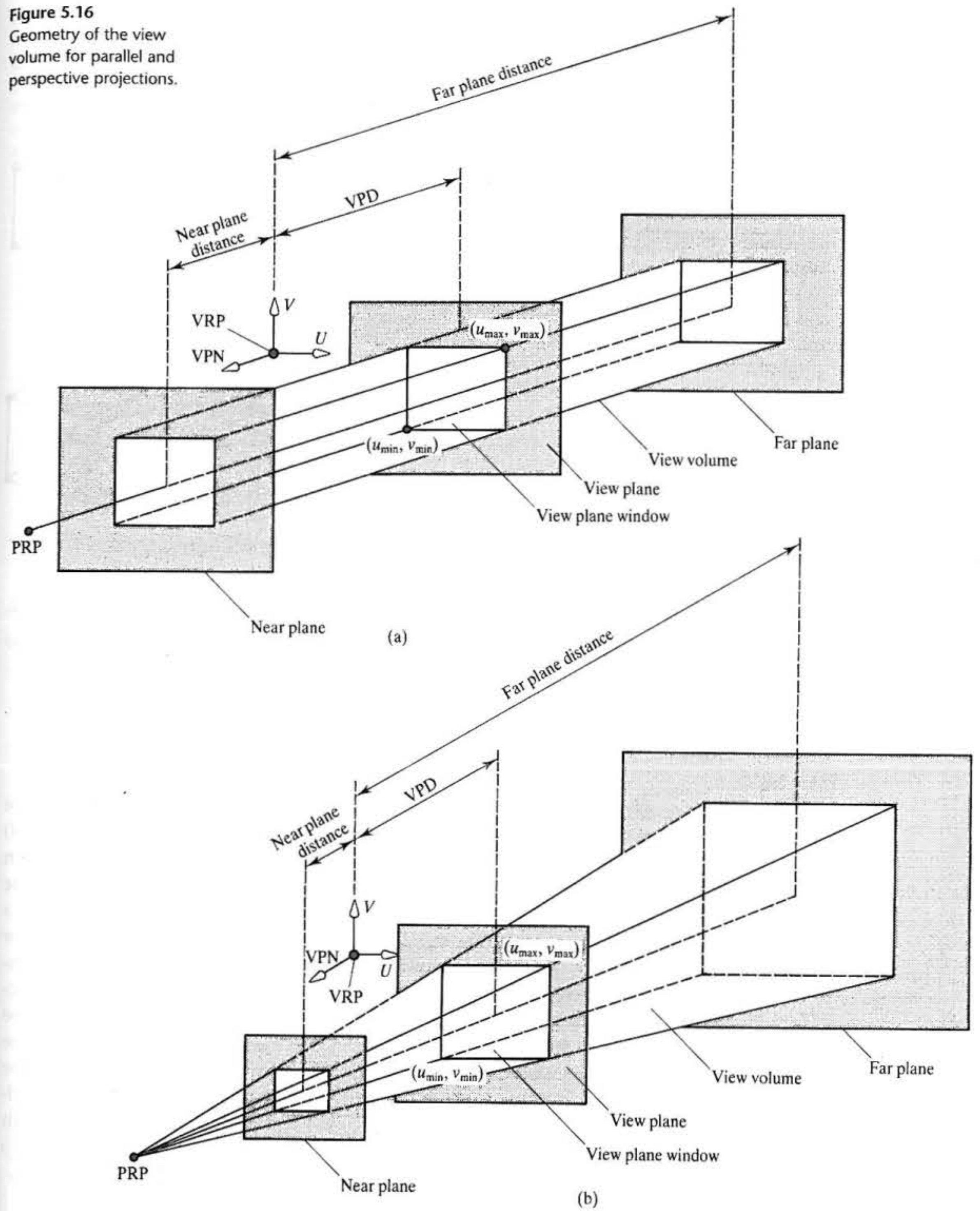
Figure 5.15
PHIGS – view orientation and view mapping parameters. Note that this is a right-handed coordinate system.

First the view plane is established. This is a plane of (theoretically) infinite extent, parallel to the *UV* plane and at a distance given by the VPD from the VRP. A view volume is established by defining a front and back plane, together with a view window. A view window is any rectangular window in the view plane. It is defined by minimax coordinates in two-dimensional view space. That is the *UV* coordinates transformed along the VPN into the view plane. These are (u_{min}, v_{min}) and (u_{max}, v_{max}). These relationships are shown for a parallel and a perspective projection in Figure 5.16.

A projection type and PRP complete the picture. The PRP as we stated above need not lie on a line, parallel to the VPN, and through the centre of the view plane window. If it lies off this line an oblique projection will be formed.

Two possible relationships between the PRP and the view plane window are shown in Figure 5.17. These are:

Figure 5.16
Geometry of the view volume for parallel and perspective projections.



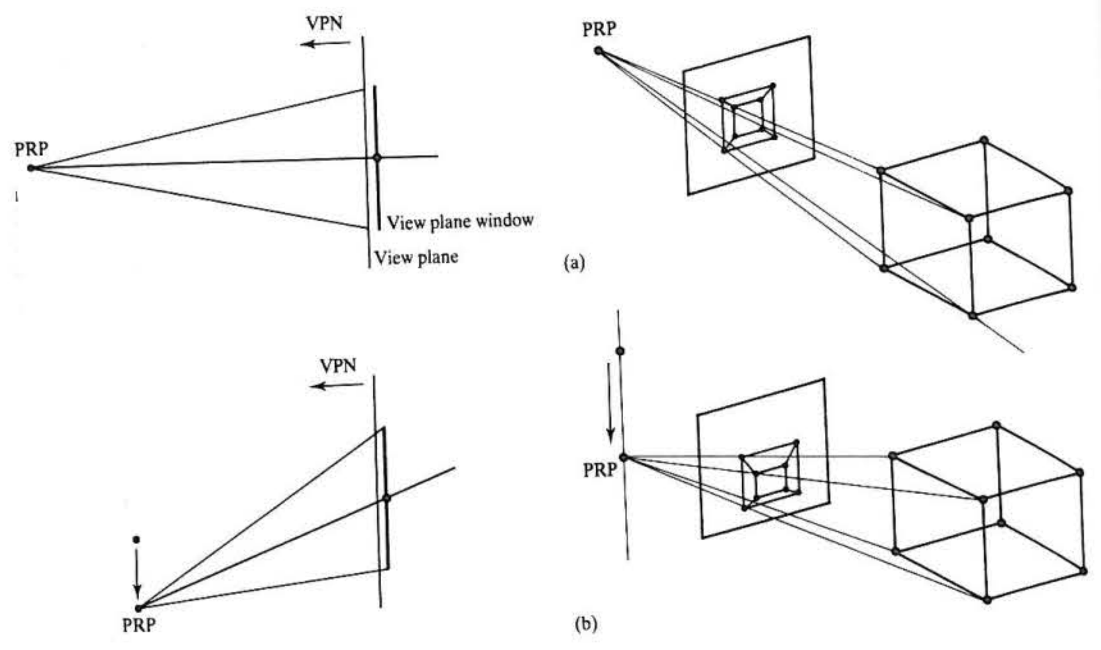


Figure 5.17
(a) 'Standard' projection and (b) oblique projection obtained by moving the PRP vertically down in a direction parallel to the view plane.

(1) A line from the PRP to the view plane window centre is parallel to the VPN.
(2) Moving the PRP results in an oblique projection. The condition in (1) is no longer true.

5.3.4 The view plane in more detail

The position and orientation of a view plane are defined by the view reference point (VRP), the view plane distance (VPD) and the view plane normal (VPN) (see Figure 5.15). The viewing direction is thus set by the VPN. Compared with the previous system, which used the camera and a focus point to establish the VPN and a distance *d*; we now use two vectors – the VRP and the VPN (plus a distance, the view plane distance (VPD) which displaces the VRP from the view plane). Unlike the previous two systems where the VRP was also used as a centre of projection the PRP or projection reference point has to be separately specified. The VRP is now just a reference point for a coordinate system and can be placed anywhere that is convenient. It can, for example, form the view plane origin, or it can be placed at the world coordinate origin, or at the centre of the object of interest. Placing it other than at the view plane origin has the advantage that the VPD has some meaning as a view distance. If the VRP is located in the view plane the VPD becomes zero and is redundant as a parameter. Also to move further or nearer to an object only the VPD needs changing.

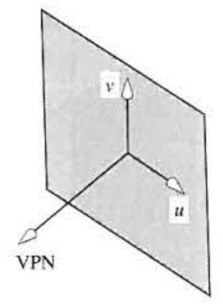


Figure 5.18
A *uv* coordinate system is established in the view plane forming a three-dimensional left-handed (right-handed for GKS-3D and PHIGS) system with the VPN.

A view plane can be positioned anywhere in world coordinate space. It can be behind, in front of, or cut through objects. Having established the position and orientation of a view plane we set up a *uv* coordinate system in the view plane with the VRP (or its projection with the view plane) as origin (Figure 5.18). The two-dimensional *uv* coordinate system and the VPN form a three-dimensional right-handed coordinate system. This enables the 'twist' of the view plane window about the VPN to be established and a window to be set up (Figure 5.19). This twist is set by the view up vector (VUV) and the general effect of this vector is to determine whether the scene is viewed upright, upside down or whatever. The direction of the *v* axis is determined by the projection of the VUV parallel to the VPN onto the view plane.

With a two-dimensional coordinate system established in the view plane a two-dimensional window can be set (Figure 5.20). This takes care of the mapping from the unconstrained or application-oriented vertex values in world coordinate space to appropriate values in the view plane extent. All other things being equal, this window setting determines the size of the object(s) on the view surface.

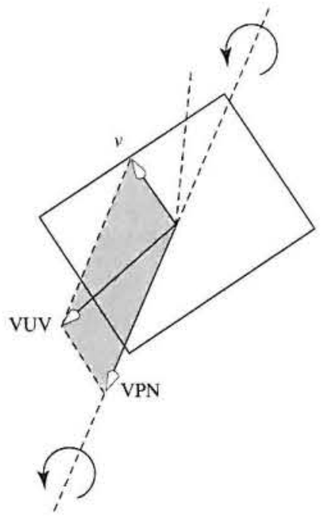


Figure 5.19
The VUV establishes the direction of the *v* axis allowing the view plane to 'twist' about the VPN.

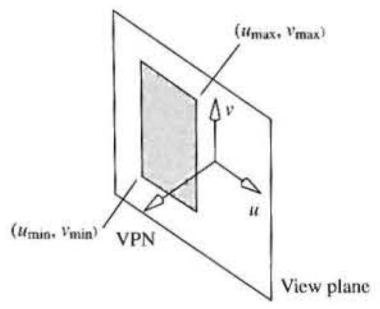


Figure 5.20
Establishing a two-dimensional window in the view plane.

5.3.5 Implementing a PHIGS-type viewing system

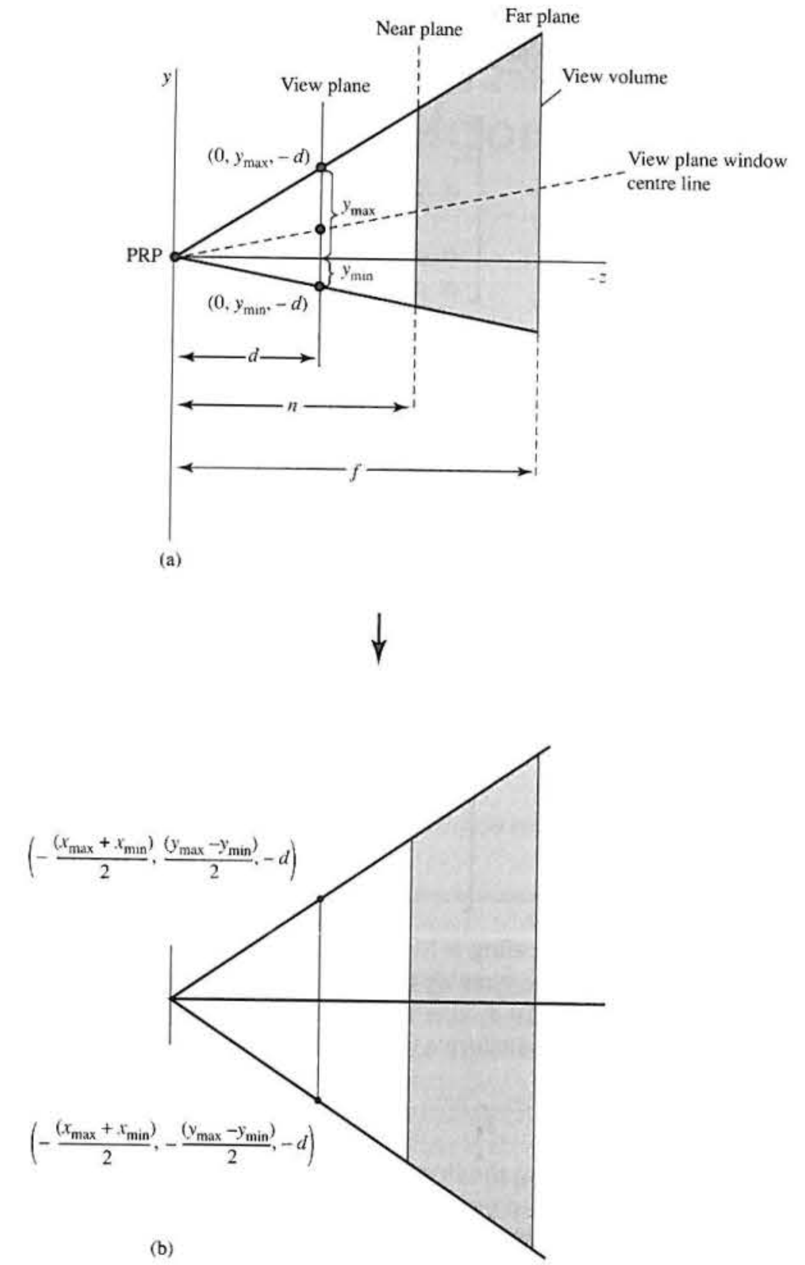
Having developed transformations for a simplified viewing system we can deal with a PHIGS-type system by extending these. First consider the PRP. In a PHIGS-type system this is specified as a point in VRC space, that is a point relative to the VRC origin. We can deal with the space disparity between the PRP and the VRC by making the PRP the origin. The view plane and clip plane parameters are distances that need to be expressed relative to the centre of projection and this is accomplished by applying the translation – PRP – to them. We can simplify the PHIGS-type viewing system parameters so that they give us the situation shown in Figure 5.21 which should be compared with Figure 5.13(a). Another feature of the PHIGS-type system is that we now have a view plane, a near plane and a far plane and the view volume has sides of different slope because we have removed the view plane window constraint of the simple system. We adopt the convention shown in Table 5.1.

(Note that although *f* and *d* have a similar interpretation to *f* and *d* of the simple system they are not the same values. We have retained the same symbols to save a proliferation in the notation.)

$\boldsymbol{T}_{\text{pers1}}$ now splits into two components and we have:

Table 5.1

Interface values	After transforming the PRP to the VRC origin
VPD	d
Far plane distance	f
Near plane distance	n
$u_{\max}, u_{\min}$	$x_{\max}, x_{\min}$
$v_{\max}, v_{\min}$	$y_{\max}, y_{\min}$



Near plane
Far plane
View plane
View volume
y
$(0, y_{\max}, -d)$
$y_{\max}$
View plane window centre line
PRP
$y_{\min}$
$-z$
$(0, y_{\min}, -d)$
d
n
f
(a)
$\left(-\frac{(x_{\max}+x_{\min})}{2}, \frac{(y_{\max}-y_{\min})}{2}, -d\right)$
$\left(-\frac{(x_{\max}+x_{\min})}{2}, -\frac{(y_{\max}-y_{\min})}{2}, -d\right)$
(b)

Figure 5.21
(a) The situation after making the PRP the origin.
(b) After transforming to a symmetrical view volume.

$$\boldsymbol{T}_{\text{pers}} = \boldsymbol{T}_{\text{pers2}}\boldsymbol{T}_{\text{pers1b}}\boldsymbol{T}_{\text{pers1a}}$$

where $\boldsymbol{T}_{\text{pers1b}}$ and $\boldsymbol{T}_{\text{pers2}}$ have the same effect as $\boldsymbol{T}_{\text{pers1}}$ and $\boldsymbol{T}_{\text{pers2}}$ of the previous viewing system. These are obtained by modifying $\boldsymbol{T}_{\text{pers1}}$ and $\boldsymbol{T}_{\text{pers2}}$ to include the view plane window parameters and the separation of the near plane from the view plane.

We first need to shear such that the view volume centre line becomes coincident with the z_v axis. This means adjusting x and y values by an amount proportional to z. It is easily seen that:

$$\boldsymbol{T}_{\text{pers1a}} = \begin{bmatrix} 1 & 0 & \dfrac{x_{\max}+x_{\min}}{2d} & 0 \\ 0 & 1 & \dfrac{y_{\max}+y_{\min}}{2d} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

For example, the upper and lower view plane window edges transform as follows:

$$(0, y_{\max}, -d, 1) \quad \text{transforms to} \quad \left(-\frac{x_{\max}+x_{\min}}{2}, \frac{y_{\max}-y_{\min}}{2}, -d, 1\right)$$

and

$$(0, y_{\min}, -d, 1) \quad \text{transforms to} \quad \left(-\frac{x_{\max}+x_{\min}}{2}, -\frac{y_{\max}-y_{\min}}{2}, -d, 1\right)$$

transforming the original view volume into a symmetrical view volume (Figure 5.21(b)).

The second transform is:

$$\boldsymbol{T}_{\text{pers1b}} = \begin{bmatrix} \dfrac{2d}{x_{\max}-x_{\min}} & 0 & 0 & 0 \\ 0 & \dfrac{2d}{y_{\max}-y_{\min}} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This scaling is identical in effect to $\boldsymbol{T}_{\text{pers1}}$ of the simple viewing system converting the symetrical view volume to a 45° view volume. For example, consider the effect of $\boldsymbol{T}_{\text{pers1}} = \boldsymbol{T}_{\text{pers1b}}\boldsymbol{T}_{\text{pers1a}}$ on the line through the view plane window centre. This transforms the point:

$$\left(\frac{x_{\max}+x_{\min}}{2}, \frac{y_{\max}+y_{\min}}{2}, -d, 1\right) \quad \text{to} \quad (0, 0, -d, 1)$$

making the line from the origin to this point coincident with the $-z$ axis – the required result.

Finally we have:

$$\boldsymbol{T}_{\text{pers2}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & f/(f-n) & -fn/(f-n) \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

which maps the regular pyramid into a box.

6 The graphics pipeline (2): rendering or algorithmic processes

Introduction

In this chapter we will describe the algorithmic operations that are required to render a polygon mesh object. We will describe a particular, but common, approach which uses a hidden surface algorithm called the Z-buffer algorithm and which utilizes some form of interpolative shading. The advantage of this strategy is that it enables us to fetch individual polygons from the object database in any order. It also means that there is absolutely no upward limit on scene complexity as far as the polygon count is concerned. The enduring success of this approach is due not only to these factors but also to the visual success of interpolative shading techniques in making the piecewise linear nature of the object almost invisible. The disadvantage of this approach, which is not without importance, is its inherent inefficiency. Polygons may be rendered to the screen which are subsequently overwritten by polygons nearer to the viewer.

In this renderer the processes that we need to perform are, rasterization, or finding the set of pixels onto which a polygon projects, hidden surface removal and shading. To this we add clipping against the view volume, a process that we dealt with briefly as a pure geometric operation in the previous chapter. In this chapter we will develop an algorithmic structure that 'encloses' the geometric operation.

As we remarked in the previous chapter, these processes are mostly carried out in three-dimensional screen space, the innermost loop in the algorithm being a 'for each' pixel structure. In other words, the algorithms are known as screen space algorithms. This is certainly not always the case – rendering by ray tracing is mostly a view space algorithm and rendering using the radiosity method is a world space algorithm.

6.1 Clipping polygons against the view volume

We have already considered the principle of clipping in the previous chapter and now we will describe an efficient structure for the task. In that analysis we saw how to determine whether a single point was within the view volume. This is an inefficient approach – we need a fast method for evaluating whether an object is wholly outside, wholly inside or straddles the view volume. Clipping has become an extremely important operation with the growth of polygon counts and the demand for real time rendering. In principle we want to discard as many polygons as possible at an early stage in the rendering pipeline. The two common approaches to avoiding detailed low-level testing are scene management techniques and bounding volumes. (Bounding volumes themselves can be considered a form of scene management.) We will look at using a simple bounding volume.

It is possible to perform a simple test that will reject objects wholly outside the view volume and accept those wholly within the view volume. This can be achieved by calculating a bounding sphere for an object and testing this against the view volume. The radius of the bounding sphere of an object is a fixed value

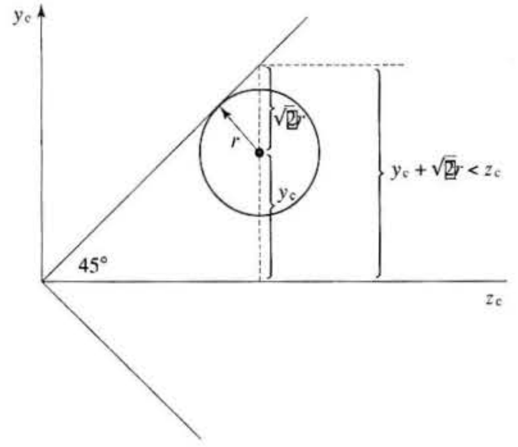


Figure 6.1
Showing one of the conditions for a bounding sphere to lie wholly within the view volume.

and this can be pre-calculated and stored as part of the database. When an object is processed by the 3D viewing pipeline, its local coordinate system origin (the bounding sphere we assume is centred on this origin) is also transformed by the pipeline.

If the object is completely outside the view volume it can be discarded; if it is entirely within the view volume it does not need to be clipped. If neither of these conditions applies then it may need to be clipped. We cannot be certain because although the sphere may intersect the view volume the object that it contains may not. This problem with bounding objects affects their use throughout computer graphics and it is further examined in Chapter 12.

For a sphere the conditions are easily shown (Figure 6.1) to be:

- **Completely inside if all following conditions are true:**

$z_c > x_c + \sqrt{2}r$

$z_c > -x_c + \sqrt{2}r$

$z_c > y_c + \sqrt{2}r$

$z_c > -y_c + \sqrt{2}r$

$z_c > r + n$

$z_c > -r + f$

- **Completely outside if any of the following conditions apply:**

$z_c > x_c - \sqrt{2}r$

$z_c > -x_c - \sqrt{2}r$

$z_c > y_c - \sqrt{2}r$

$z_c > -y_c - \sqrt{2}r$

$z_c > -r + n$

$z_c > r + f$

where:

$$\begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} = \begin{bmatrix} d/h & 0 & 0 & 0 \\ 0 & d/h & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix}$$

In other words, this operation takes place in the clipping space shown in Figure 5.13.

Objects that need to be clipped can be dealt with by the Sutherland–Hodgman re-entrant polygon clipper (Sutherland and Hodgman 1974). This is easily extended to three dimensions. A polygon is tested against a clip boundary by testing each polygon edge against a single infinite clip boundary. This structure is shown in Figure 6.2.

We consider the innermost loop of the algorithm, where a single edge is being tested against a single clip boundary. In this step the process outputs zero, one or two vertices to add to the list of vertices defining the clipped polygon. Figure 6.3

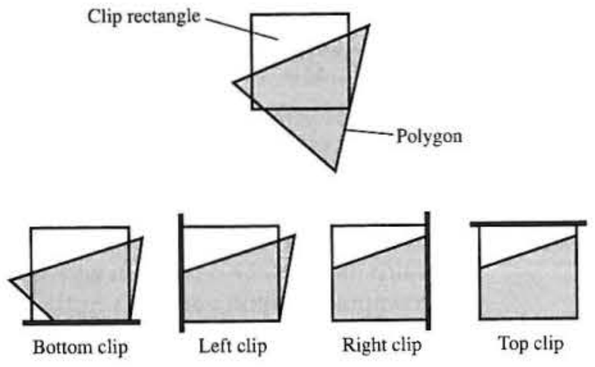


Figure 6.2
Sutherland–Hodgman clipper clips each polygon against each edge of each clip rectangle.

shows the four possible cases. An edge is defined by vertices ***S*** and ***F***. In the first case the edge is inside the clip boundary and the existing vertex ***F*** is added to the output list. In the second case the edge crosses the clip boundary and a new vertex ***I*** is calculated and output. The third case shows an edge that is completely outside the clip boundary. This produces no output. (The intersection for the edge that caused the excursion outside is calculated in the previous iteration and the intersection for the edge that causes the incursion inside is calculated in the next iteration.) The final case again produces a new vertex which is added to the output list.

To calculate whether a point or vertex is inside, outside or on the clip boundary we use a dot product test. Figure 6.4 shows a clip boundary ***C*** with an outward normal $\boldsymbol{N}_c$ and a line with end points ***S*** and ***F***. We represent the line parametrically as:

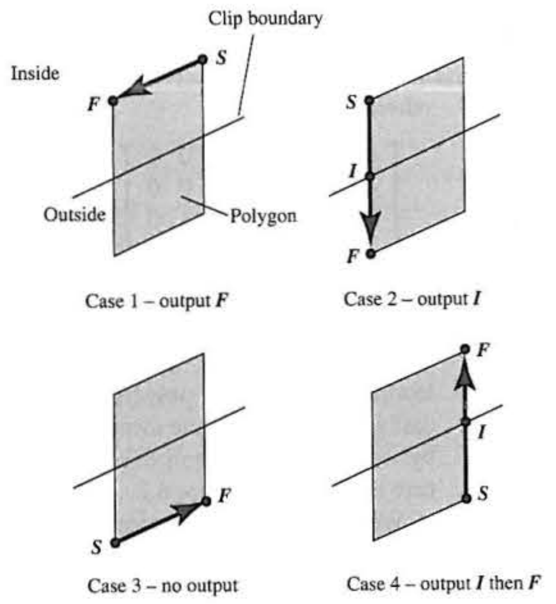
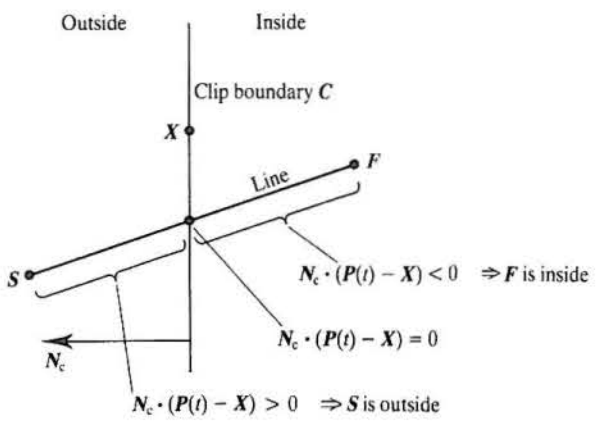


Figure 6.3
Sutherland–Hodgman clipper – within the polygon loop each edge of a polygon is tested against each clip boundary.

Figure 6.4
Dot product test to determine whether a line is inside or outside a clip boundary.


Outside
Inside
Clip boundary C
X
F
Line
S
N_c
N_c · (P(t) – X) < 0 ⇒ F is inside
N_c · (P(t) – X) = 0
N_c · (P(t) – X) > 0 ⇒ S is outside

$$\boldsymbol{P}(t) = \boldsymbol{S} + (\boldsymbol{F} - \boldsymbol{S})t \qquad [6.1]$$

where:

$0 \le t \le 1$

We define an arbitrary point on the clip boundary as ***X*** and a vector from ***X*** to any point on the line. The dot product of this vector and the normal allows us to distinguish whether a point on the line is outside, inside or on the clip boundary. In the case shown in Figure 6.4:

$\boldsymbol{N}_c\cdot(\boldsymbol{S} - \boldsymbol{X}) > 0 \quad \Rightarrow \boldsymbol{S}$ is outside the clip region

$\boldsymbol{N}_c\cdot(\boldsymbol{F} - \boldsymbol{X}) < 0 \quad \Rightarrow \boldsymbol{F}$ is inside the clip region

and:

$\boldsymbol{N}_c\cdot(\boldsymbol{P}(t) - \boldsymbol{X}) = 0$

defines the point of intersection of the line and the clip boundary. Solving Equation 6.1 for *t* enables the intersecting vertex to be calculated and added to the output list.

In practice the algorithm is written recursively. As soon as a vertex is output the procedure calls itself with that vertex and no intermediate storage is required for the partially clipped polygon. This structure makes the algorithm eminently suitable for hardware implementation.

6.2 Shading pixels

The first quality shading in computer graphics was developed by H. Gouraud in 1971 (Gouraud 1971). In 1975 Phong Bui-Tuong (Phong 1975) improved on Gouraud's model and Phong shading, as it is universally known, became the *de facto* standard in mainstream 3D graphics. Despite the subsequent development of 'global' techniques, such as ray tracing and radiosity, Phong shading has remained ubiquitous. This is because it enables reality to be mimicked to an acceptable level at reasonable cost.

There are two separate considerations to shading the pixels onto which a polygon projects. First we consider how to calculate the light reflected at any point on the surface of an object. Given a theoretical framework that enables us to do this, we can then calculate the light intensity at the pixels onto which the polygon projects. The first consideration we call 'local reflection models' and the second 'shading algorithms'. The difference is illustrated conceptually in Figure 6.5. For example, one of the easiest approaches to shading – Gouraud shading – applies a local reflection model at each of the vertices to calculate a vertex intensity, then derives a pixel intensity using the same interpolation equations as we used in the previous section to interpolate depth values.

Basically there is a conflict here. We only want to calculate the shade for each pixel onto which the polygon projects. But the reflected light intensity at every point on the surface of a polygon is by definition a world space calculation. We are basing the calculation on the orientation of the surface with respect to a light source both of which are defined in world space. Thus we use a 2D projection of the polygon as the basis of an interpolation scheme that controls the world space calculations of intensity and this is incorrect. Linear interpolation, using equal increments, in screen space does not correspond to how the reflected intensity should vary across the face of the polygon in world space. One of the reasons for this is that we have already performed a (non-affine) perspective transformation to get into screen space. Like many algorithms in 3D computer graphics it produces an acceptable visual result, even using incorrect mathematics. However, this approach does lead to visible artefacts in certain contexts. The comparative study in Chapter 18 has an illustration of an artefact caused by this.

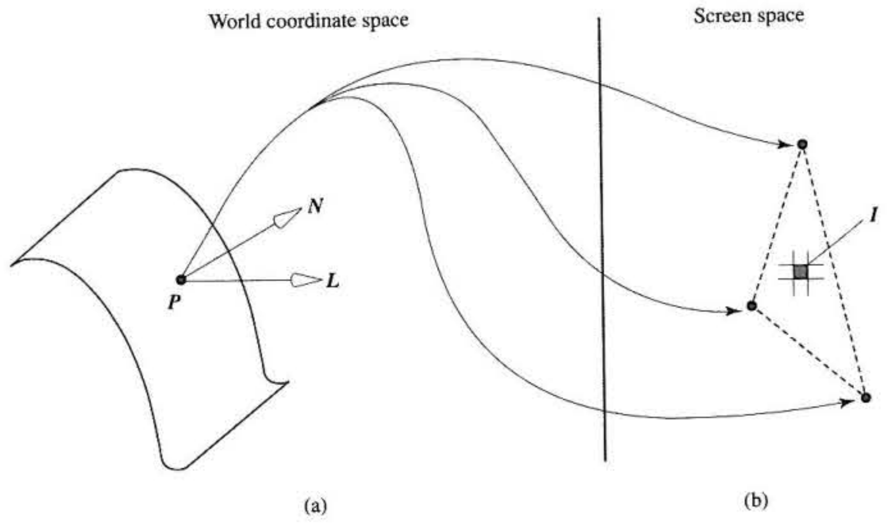


Figure 6.5
Illustrating the difference between local reflection models and shading algorithms. (a) Local reflection models calculate light intensity at any point ***P*** on the surface of an object. (b) Shading algorithms interpolate pixel values from calculated light intensities at the polygon vertices.

6.2.1 Local reflection models

A local reflection model enables the calculation of the reflected light intensity from a point on the surface of an object. The development of a variety of local reflection models is dealt with in Chapter 7, here we will confine ourselves to considering, from a practical view point, the most common model and see how it fits into a renderer.

This model, introduced in 1975, evaluates the intensity of the reflected light as a function of the orientation of the surface at the point of interest with respect to the position of a point light source and surface properties. We refer to such a model as a local reflection model because it only considers direct illumination. It is as if the object under consideration was an isolated object floating in free space. Interaction with other objects that result in shadows and inter-reflection are not taken into account by local reflection models. This point is emphasized in Figure 6.6; in Chapter 10 we deal with global illumination in detail.

The physical reflection phenomena that the model simulates are:

- Perfect specular reflection.
- Imperfect specular reflection.
- Perfect diffuse reflection.

These are illustrated in Figure 6.7 for a point light source that is sending an infinitely thin beam of light to a point on a surface. Perfect specular reflection occurs when incident light is reflected, without diverging, in the 'mirror' direction. Imperfect specular reflection is that which occurs when a thin beam of light strikes an imperfect mirror, that is a surface whose reflecting properties are that of a perfect mirror but only at a microscopic level – because the surface is physically rough. Any area element of such a surface can be considered to be made up of thousands of tiny perfect mirrors all at slightly different orientations.

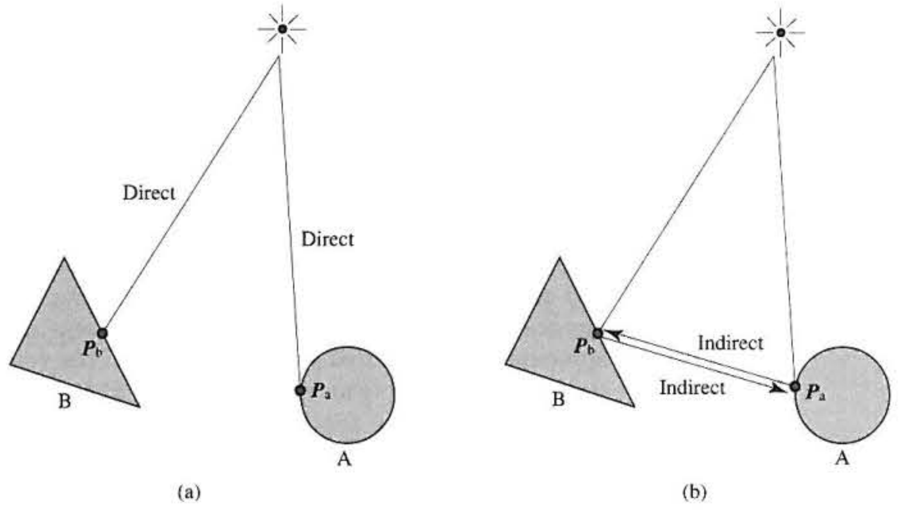


Figure 6.6
(a) A local reflection model calculates intensity at P_b and P_a considering direct illumination only. (b) Any indirect reflected light from A to B or from B to A is not taken into account.

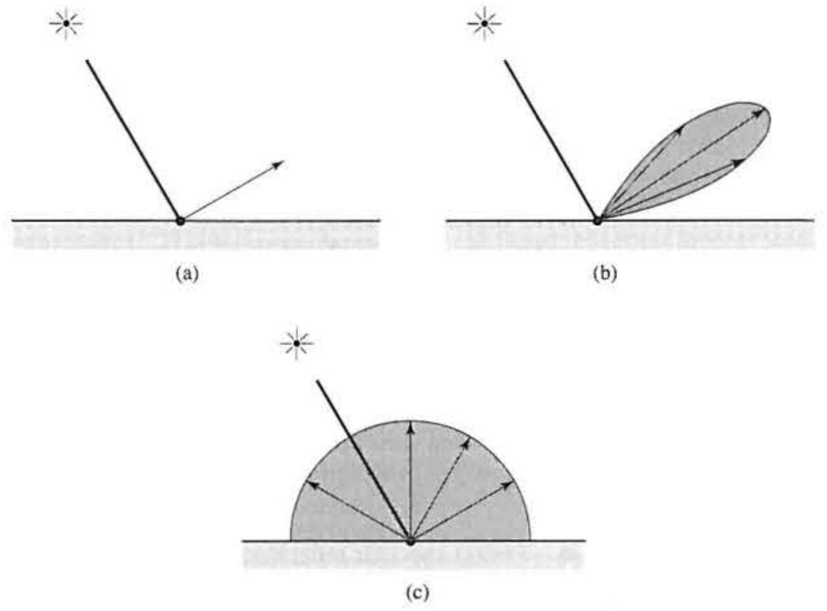


Figure 6.7
The three reflection phenomena used in computer graphics.
(a) Perfect specular reflection. (b) Imperfect specular reflection.
(c) Perfect diffuse reflection.

Perfect specular reflection does not occur in practice but we use it in ray tracing models (see Chapter 12) simply because calculating interaction due to imperfect specular reflection is too expensive. A perfect diffuse surface reflects the light equally in all directions and such a surface is usually called matte.

The Phong reflection model considers the reflection from a surface to consist of three components linearly combined:

reflected light = ambient light + diffuse component + specular component

The ambient term is a constant and simulates global or indirect illumination. This term is necessary because parts of a surface that cannot 'see' the light source, but which can be seen by the viewer, need to be lit. Otherwise they would be rendered as black. In reality such lighting comes from global or indirect illumination and simply adding a constant side-steps the complexity of indirect or global illumination calculations.

It is useful to consider what types of surface such a model simulates. Linear combination of a diffuse and specular component occurs in polished surfaces such as varnished wood. Specular reflection results from the transparent layer and diffuse reflection from the underlying surface (Figure 6.8). Many different physical types, although not physically the same as a varnished wood, can be approximately simulated by the same model. The veracity of this can be demonstrated by considering looking at a sample of real varnished wood, shiny plastic and gloss paint. If all contextual clues are removed and the reflected light from each sample exhibited the same spectral distribution, an observer would find it difficult to distinguish between the samples.

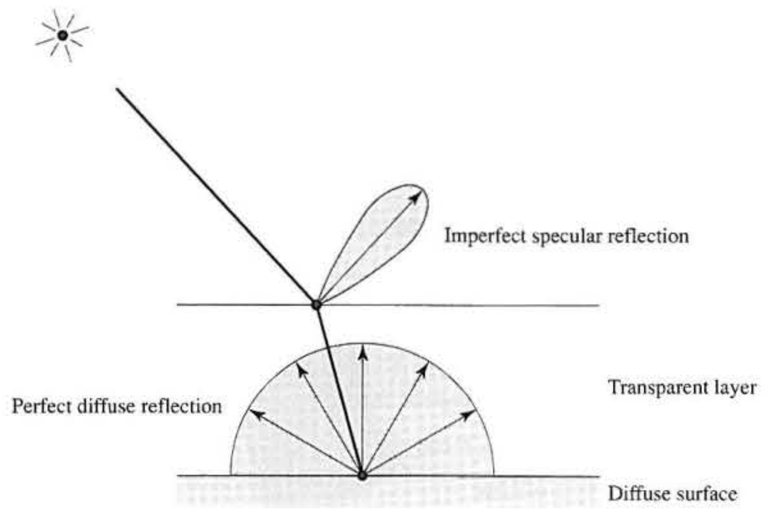


Figure 6.8
The 'computer graphics' surface.

As well as possessing the limitation of being a local model, the Phong reflection model is completely empirical or imitative. One of its major defects is that the value of reflected intensity calculated by the model is a function only of the viewing direction and the orientation of the surface with respect to the light source. In practice, reflected light intensity exhibits bi-directional behaviour. It depends also on the direction of the incident light. This defect has led to much research into physically based reflection models, where an attempt is made to model reflected light by simulating real surface properties. However, the subtle improvements possible by using such models – such as the ability to make surfaces look metallic – have not resulted in the demise of the Phong reflection model and the main thrust of current research into rendering methods deals with the limitation of 'localness'. Global methods, such as radiosity, result in much more significant improvements to the apparent reality of a scene.

Leaving aside, for a moment, the issue of colour, the physical nature of a surface is simulated by controlling the proportion of the diffuse to specular reflection and we have the reflected light:

$$I = k_aI_a + k_dI_d + k_sI_s$$

Where the proportions of the three components, ambient, diffuse and specular are controlled by three constants, where:

$$k_a + k_d + k_s = 1$$

Consider I_d. This is evaluated as:

$$I_d = I_i \cos \theta$$

where:

I_i is the intensity of the incident light

θ is the angle between the surface normal at the point of interest and the direction of the light source

In vector notation:

$I_d = I_i\,(\boldsymbol{L}\cdot\boldsymbol{N})$

The geometry is shown in Figure 6.9.

Now physically the specular reflection consists of an image of the light source 'smeared' across an area of the surface resulting in what is commonly known as a highlight. A highlight is only seen by a viewer if the viewing direction is near to the mirror direction. We therefore simulate specular reflection by:

$I_s = I_i \cos^n \Omega$

where:

Ω is the angle between the viewing direction and the mirror direction $\boldsymbol{R}$

n is an index that simulates the degree of imperfection of a surface

When $n = \infty$ the surface is a perfect mirror – all reflected light emerges along the mirror direction. For other values of n an imperfect specular reflector is simulated (Figure 6.7(b)). The geometry of this is shown in Figure 6.10. In vector notation we have:

$I_s = I_i\,(\boldsymbol{R}\cdot\boldsymbol{V})^n$

Bringing these terms together gives:

$I = k_a I_a + I_i(k_d(\boldsymbol{L}\cdot\boldsymbol{N}) + k_s(\boldsymbol{R}\cdot\boldsymbol{V})^n)$

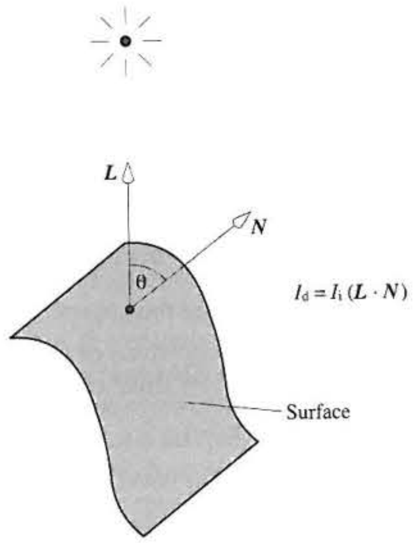


Figure 6.9
The Phong diffuse component.

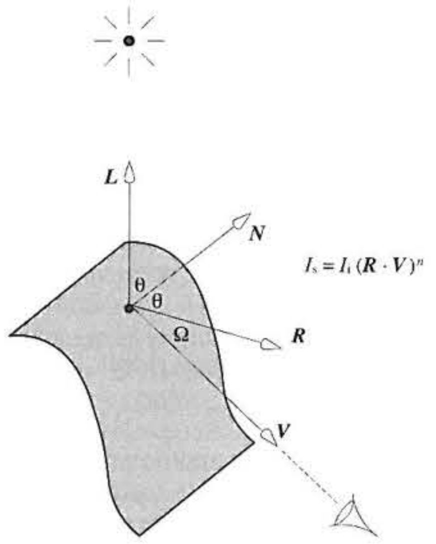


Figure 6.10
The Phong specular component.

The behaviour of this equation is illustrated in Figures 6.11 and 6.12 (Colour Plate). Figure 6.11 shows the light intensity at a single point $\boldsymbol{P}$ as a function of the orientation of the viewing vector $\boldsymbol{V}$. The semicircle is the sum of the constant ambient term and the diffuse term – which is constant for a particular value of $\boldsymbol{N}$. Addition of the specular term gives the profile shown in the figure. As the value of n is increased the specular bump is narrowed. Figure 6.12 (Colour Plate) shows the equation applied to the same object using different values of k_s and k_d and n.

6.2.2 Local reflection models – practical points

A number of practical matters that deal with colour and the simplification of the geometry now need to be explained.

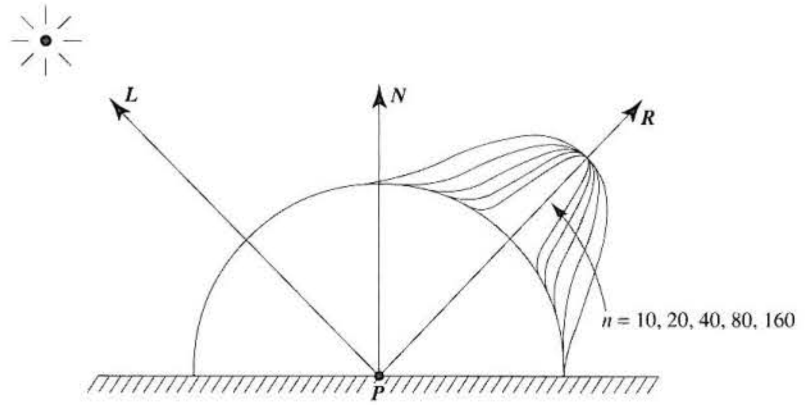


Figure 6.11
The light intensity at point $\boldsymbol{P}$ as a function of the orientation of the viewing vector $\boldsymbol{V}$.

The expense of the above shading equation, which is applied a number of times at every pixel, can be considerably reduced by making geometric simplifications that reduce the calculation time, but which do not affect the quality of the shading. First if the light source is considered as a point source located at infinity then $\boldsymbol{L}$ is constant over the domain of the scene. Second we can also place the view point at infinity making $\boldsymbol{V}$ constant. Of course, for the view and perspective transformation, the view point needs to be firmly located in world space so we end up using a finite view point for the geometric transformations and an infinite one for the shading equation.

Next the vector $\boldsymbol{R}$ is expensive to calculate and it is easier to define a vector $\boldsymbol{H}$ (halfway) which is the unit normal to a hypothetical surface that is oriented in a direction halfway between the light direction vector $\boldsymbol{L}$ and the viewing vector $\boldsymbol{V}$ (Figure 6.13). It is easily seen that:

$$\boldsymbol{H} = (\boldsymbol{L} + \boldsymbol{V})/2$$

This is the orientation that a surface would require if it was to reflect light maximally along the $\boldsymbol{V}$ direction. Our shading equation now becomes:

$$I = I_a k_a + I_i(k_d\,(\boldsymbol{L}\cdot\boldsymbol{N}) + (\boldsymbol{N}\cdot\boldsymbol{H})^n)$$

because the term $(\boldsymbol{N}\cdot\boldsymbol{H})$ varies in the same manner as $(\boldsymbol{R}\cdot\boldsymbol{V})$. These simplifications mean that I is now a function only of $\boldsymbol{N}$.

For coloured objects we generate three components of the intensity I_r, I_g and I_b controlling the colour of the objects by appropriate setting of the diffuse reflection coefficients k_r, k_b and k_g. In effect the specular highlight is just the reflection of the light source in the surface of the object and we set the proportions of the k_s to match the colour of the light. For a white light, k_s is equal in all three equations. Thus we have:

$$I_r = I_a k_{ar} + I_i((k_{dr}(\boldsymbol{L}\cdot\boldsymbol{N}) + k_s(\boldsymbol{N}\cdot\boldsymbol{H})^n)$$
$$I_g = I_a k_{ag} + I_i((k_{dg}(\boldsymbol{L}\cdot\boldsymbol{N}) + k_s(\boldsymbol{N}\cdot\boldsymbol{H})^n)$$
$$I_b = I_a k_{ab} + I_i((k_{db}(\boldsymbol{L}\cdot\boldsymbol{N}) + k_s(\boldsymbol{N}\cdot\boldsymbol{H})^n)$$

A more detailed treatment of colour in computer graphics is given in Chapter 15.

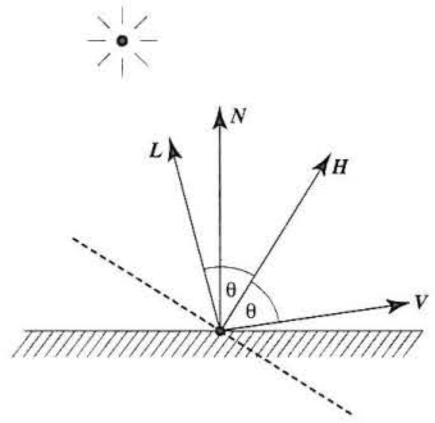
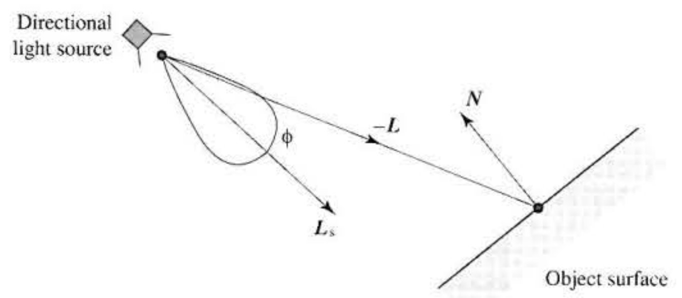


Figure 6.13
H is the normal to a surface orientation that would reflect all the light along ***V***.

Figure 6.14
Light source represented as a specularly reflecting surface.



6.2.3 Local reflection models – light source considerations

One of the most limiting approximations in the above model is reducing the light source to a point at infinity. Also we can see in Figure 6.12 (Colour Plate) that there is an unsatisfactory confusion concerning the interpretation of the parameter n that is supposed to give the impression of 'glossiness'. In practice it looks as if we are changing the size of the light source.

A simple directional light (non-point) source is easily modelled and the following was suggested by Warn (1983). In this method a directional light source is modelled in the same way as a specularly reflecting surface, where the light emitted from the source is given by a cosine function raised to a power. Here we assume that for a directional source, the light intensity in a particular direction, given by the angle ϕ is:

$$I_s \cos^m\phi$$

Now ϕ is the angle between $-\boldsymbol{L}$, the direction of the point on the surface that we are considering, and $\boldsymbol{L}_s$, the orientation of the light source (Figure 6.14). The value of I_i that we use in the shading equation is then given by:

$$I_i = I_s(-\boldsymbol{L}\cdot\boldsymbol{L}_s)^m$$

Note that we can no longer consider the vector $\boldsymbol{L}$ constant over the scene.

6.3 Interpolative shading techniques

Having dealt with the problem of calculating light intensity at a point, we now consider how to apply such a model to a polygon and calculate the light intensity over its surface. Two classic techniques have emerged – Gouraud and Phong shading. The difference in quality between these two techniques is shown and discussed in the comparitive case study in Chapter 18 and we now deal with each separately. Phong interpolation gives the more accurate highlights – as we shall see – and is generally the preferred model. Gouraud shading on the other hand is considerably cheaper. Both techniques have been developed both to interpolate information efficiently across the face of a polygon and to diminish the visibility of the polygon edges in the final shaded image. Information is interpolated from values at the vertices of a polygon and the situation is exactly analogous to depth interpolation.

6.3.1 Interpolative shading techniques – Gouraud shading

In Gouraud shading we calculate light intensity – using the local reflection model of the previous section – at the vertices of the polygon and then interpolate between these intensities to find values at projected pixels. To do this we use the bilinear interpolation equations given in Section 1.5, the property p being the vertex intensity I. The particular surface normals used at a vertex are special normals called vertex normals. If we consider a polygon in isolation then, of course, the vertex normals are all parallel. However, in Gouraud shading we use special normals called vertex normals and it is this device that reduces the visibility of polygon edges. Consider Figure 6.15. Here the vertex normal $\boldsymbol{N}_A$ is calculated by averaging $\boldsymbol{N}_1$, $\boldsymbol{N}_2$, $\boldsymbol{N}_3$ and $\boldsymbol{N}_4$.

$$\boldsymbol{N}_A = \boldsymbol{N}_1 + \boldsymbol{N}_2 + \boldsymbol{N}_3 + \boldsymbol{N}_4$$

$\boldsymbol{N}_A$ is then used to calculate an intensity at vertex A that is common to all the polygons that share vertex A.

For computational efficiency the interpolation equations are implemented as incremental calculations. This is particularly important in the case of the third equation, which is evaluated for every pixel. If we define Δx to be the incremental distance along a scan line then ΔI, the change in intensity from one pixel to the next, is:

$$\Delta I_s = \frac{\Delta x}{x_b - x_a}(I_b - I_a)$$

$$I_{s,n} = I_{s,n-1} + \Delta I_s$$

Because the intensity is only calculated at vertices the method cannot adequately deal with highlights and this is its major disadvantage. The cause of this defect can be understood by examining Figure 6.16(a). We have to bear in mind that the polygon mesh is an approximation to a curved surface. For a particular viewing and light source direction we can have a diffuse component at A and B and a specular highlight confined to some region between them. Clearly if we are deriving the intensity at pixel P from information at A and B we will not calculate a highlight. This situation is neatly taken care of by interpolating vertex normals rather than intensities as shown in Figure 6.16(b). This approach is known as Phong shading.

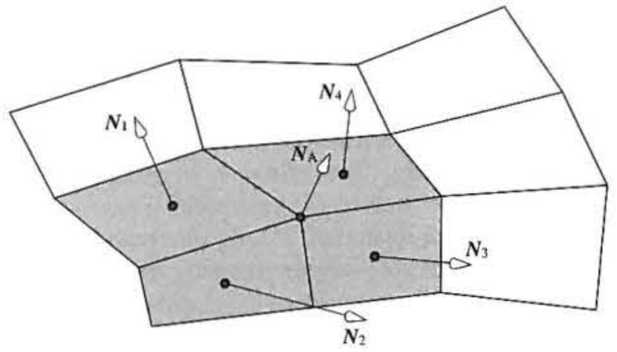


Figure 6.15
The vertex normal $\boldsymbol{N}_A$ is the average of the normals $\boldsymbol{N}_1$, $\boldsymbol{N}_2$, $\boldsymbol{N}_3$, and $\boldsymbol{N}_4$, the normals of the polygon that meet at the vertex.

6.3.2 Interpolative shading techniques – Phong shading

Here we interpolate vertex normals across the polygon interior and calculate for each polygon pixel projection an interpolated normal. This interpolated normal is then used in the shading equation which is applied for every pixel projection. This has the geometric effect (Figure 6.16) of 'restoring' some curvature to polygonally faceted surfaces.

The price that we pay for this improved model is efficiency. Not only is the vector interpolation three times the cost of intensity interpolation, but each vector has to be normalized and a shading equation calculated for each pixel projection.

Incremental computations can be employed as with intensity interpolation, and the interpolation would be implemented as:

$$N_{sx,n} = N_{sx,n-1} + \Delta N_{sx}$$
$$N_{sy,n} = N_{sy,n-1} + \Delta N_{sy}$$
$$N_{sz,n} = N_{sz,n-1} + \Delta N_{sz}$$

Where N_{sx}, N_{sy} and N_{sz} are the components of a general scan line normal vector $\boldsymbol{N}_s$ and:

$$\Delta N_{sx} = \frac{\Delta x}{x_b - x_a}(N_{bx} - N_{ax})$$

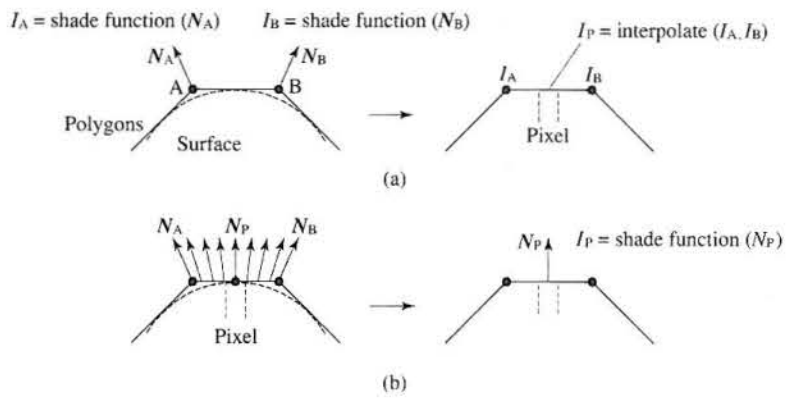

I_A = shade function (N_A)
I_B = shade function (N_B)
I_P = interpolate (I_A, I_B)
N_A
N_B
A
B
Polygons
Surface
I_A
I_B
Pixel
(a)
N_A
N_P
N_B
Pixel
N_P
I_P = shade function (N_P)
(b)

Figure 6.16
Illustrating the difference between Gouraud and Phong shading. (a) Gouraud shading. (b) Phong shading.

$$\Delta N_{sy} = \frac{\Delta x}{x_b - x_a}(N_{by} - N_{ay})$$

$$\Delta N_{sz} = \frac{\Delta x}{x_b - x_a}(N_{bz} - N_{az})$$

6.3.3 Renderer shading options

Most renderers have a hierarchy of shading options where you trade wait time against the quality of the shaded image. This approach also, of course, applies to the addition of shadows and texture. The normal hierarchy is:

- **Wireframe** No rendering or shading at all. A wireframe display may be used to position objects in a scene by interacting with the viewing parameters. It is also commonly used in animation systems where an animator may be creating movement of objects in a scene interactively. He can adjust various aspects of the animation and generate a real time animation sequence in wireframe display mode. In both these applications a full shaded image is obviously not necessary. One practical problem is that using the same overall renderer strategy for wireframe rendering as for shading (that is, independently drawing each polygon) will result in each edge being drawn twice – doubling the draw time for an object.
- **Flat shaded polygons** Again a fast option. The single 'true' polygon normal is used, one shade calculated using the Gouraud equation for each polygon and the shading interpolative process is avoided.
- **Gouraud shading** The basic shading option which produces a variation across the face of polygons. Because it cannot deal properly with specular highlights, a Gouraud shading option normally only calculates diffuse reflection.
- **Phong shading** The 'standard' quality shading method which due to the vector interpolation and the evaluation of a shading equation at every pixel is between four and five times slower than Gouraud shading.
- **Mixing Phong and Gouraud shading** Consider a diffuse object. Although using Gouraud shading for the object produces a slightly different effect from using Phong shading, with the specular reflection coefficient set to zero the difference is not visually important. This suggests that in a scene consisting of specular and diffuse objects we can use Gouraud shading for the diffuse objects and only use Phong shading for the specular ones. The Gouraud–Phong option then becomes part of the object property data.

These options are compared in some detail in the comparative case study in Chapter 18.

6.3.4 Comparison of Gouraud and Phong shading

Gouraud shading is effective for shading surfaces which reflect light diffusely. Specular reflections can be modelled using Gouraud shading, but the shape of the specular highlight produced is dependent on the relative positions of the underlying polygons. The advantage of Gouraud shading is that it is computationally the less expensive of the two models, requiring only the evaluation of the intensity equation at the polygon vertices, and then bilinear interpolation of these values for each pixel.

Phong shading produces highlights which are much less dependent on the underlying polygons. But, more calculations are required involving the interpolation of the surface normal and the evaluation of the intensity function for each pixel. These facts suggest a straightforward approach to speeding up Phong shading by combining Gouraud and Phong shading.

6.4 Rasterization

Having looked at how general points within a polygon can be assigned intensities that are determined from vertex values, we now look at how we determine the actual pixels which we require intensity values for. The process is known as rasterization or scan conversion. We consider this somewhat tricky problem in two parts. First, how do we determine the pixels which the edge of a polygon straddles? Second, how do we organize this information to determine the interior points?

6.4.1 Rasterizing edges

There are two different ways of rasterizing an edge, based on whether line drawing or solid area filling is being used. Line drawing is not covered in this book, since we are interested in solid objects. However, the main feature of line drawing algorithms (for example, Bresenham's algorithm (Bresenham 1965)) is that they must generate a linear sequence of pixels with no gaps (Figure 6.17).

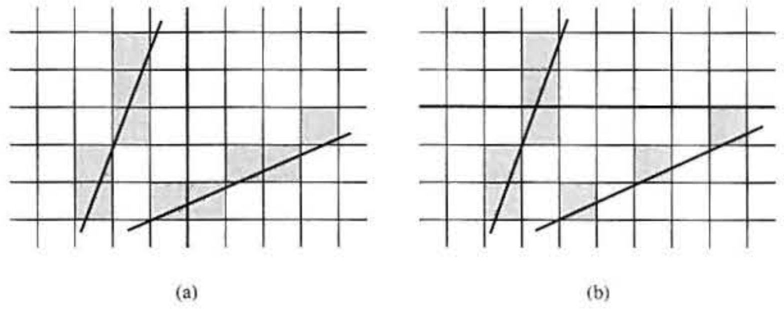


Figure 6.17
Pixel sequences required for (a) line drawing and (b) polygon filling.

For solid area filling, a less rigorous approach suffices. We can fill a polygon using horizontal line segments; these can be thought of as the intersection of the polygon with a particular scan line. Thus, for any given scan line, what is required is the left- and right-hand limits of the segment, that is the intersections of the scan line with the left- and right-hand polygon edges. This means that for each edge, we need to generate a sequence of pixels corresponding to the edge's intersections with the scan lines (Figure 6.17(b)). This sequence may have gaps, when interpreted as a line, as shown by the right-hand edge in the diagram.

The conventional way of calculating these pixel coordinates is by use of what is grandly referred to as a 'digital differential analyzer', or DDA for short. All this really consists of is finding how much the x coordinate increases per scan line, and then repeatedly adding this increment.

Let (x_s, y_s), (x_e, y_e) be the start and end points of the edge (we assume that $y_e > y_s$) The simplest algorithm for rasterizing sufficient for polygon edges is:

```
$x := x_s$
$m := (x_e - x_s)/(y_e - y_s)$
**for** $y := y_s$ **to** $y_e$ **do**
  output(round(x), y)
  $x := x + m$
```

The main drawback of this approach is that m and x need to be represented as floating point values, with a floating point addition and real-to-integer version each time round the loop. A method due to Swanson and Thayer (Swanson and Thayer 1986) provides an integer-only version of this algorithm. It can be derived from the above in two logical stages. First we separate out x and m into integer and fractional parts. Then each time round the loop, we separately add the two parts, adding a carry to the integer part should the fractional part overflow. Also, we initially set the fractional part of x to –0.5 to make rounding easy, as well as simplifying the overflow condition. In pseudocode:

```
$xi := x_s$
$xf := -0.5$
$mi := (x_e - x_s)$ **div** $(y_e - y_s)$
$mf := (x_e - x_s) / (y_e - y_s) - mi$

**for** $y := y_s$ **to** $y_e$ **do**
  output(xi,y)
  $xi := xi + mi$
  $xf := xf + mf$
  **if** $xf > 0.0$ **then** $\{xi := xi + 1;\ xf := xf - 1.0\}$
```

Because the fractional part is now independent of the integer part, it is possible to scale it throughout by $2(y_e - y_s)$, with the effect of converting everything to integer arithmetic:

```
$xi := x_s$
$xf := -(y_e - y_s)$
$mi := (x_e - x_s)$ **div** $(y_e - y_s)$
$mf := 2*[(x_e - x_s)$ **mod**$(y_e - y_s)]$

**for** $y := y_s$ **to** $y_e$ **do**
  output(xi, y)
  $xi := xi + mi$
  $xf := xf + mf$
  **if** $xf > 0$ **then** $\{xi := xi + 1;\ xf := xf - 2(y_e - y_s)\}$
```

Although this appears now to involve two divisions rather than one, they are both integer rather than floating point. Also, given suitable hardware, they can both be evaluated from the same division, since the second (**mod**) is simply the remainder from the first (**div**). Finally it only remains to point out that the $2(y_e - y_s)$ within the loop is constant and would in practice be evaluated just once outside it.

6.4.2 Rasterizing polygons

Now that we know how to find pixels along the polygon edges, it is necessary to turn our attention to filling the polygons themselves. Since we are concerned with shading, 'filling a polygon' means finding the pixel coordinates of interior points and assigning to these a value calculated using one of the incremental shading schemes described in Section 6.3. We need to generate pairs of segment end points and fill in horizontally between them. This is usually achieved by constructing an 'edge list' for each polygon.

In principle this is done using an array of linked lists, with an element for each scan line. Initially all the elements are set to NIL. Then each edge of the polygon is rasterized in turn, and the x coordinate of each pixel (x, y) thus generated is inserted into the linked list corresponding to that value of y. Each of the linked lists is then sorted in order of increasing x. The result is something like that shown in Figure 6.18. Filling-in of the polygon is then achieved by, for each scan line, taking successive pairs of x values and filling in between them (because

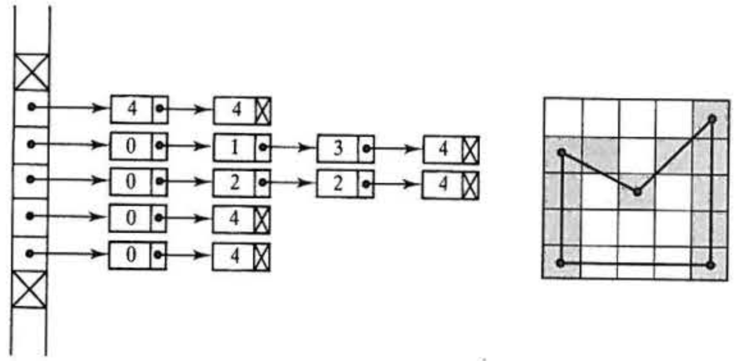


Figure 6.18
An example of a linked list maintained in ploygon rasterization.

a polygon has to be closed, there will always be an even number of elements in the linked list). Note that this method is powerful enough to cope with concave polygons with holes.

In practice, the sorting of the linked lists is achieved by inserting values in the appropriate place initially, rather than by a big sort at the end. Also, as well as calculating the x value and storing it for each pixel on an edge, the appropriate shading values would be calculated and stored at the same time (for example, intensity value for Gouraud shading; x, y and z components of the interpolated normal vector for Phong shading).

If the object contains only convex polygons then the linked x lists will only ever contain two x coordinates; the data structure of the edge list is simplified and there is no sort required. It is not a great restriction in practical computer graphics to constrain an object to convex polygons.

One thing that has been slightly glossed over so far is the consideration of exactly where the borders of a polygon lie. This can manifest itself in adjacent polygons either by gaps appearing between them, or by them overlapping. For example, in Figure 6.19, the width of the polygon is 3 units, so it should have an area of 9 units, whereas it has been rendered with an area of 16 units. The traditional solution to this problem, and the one usually advocated in textbooks, is to consider the sample point of the pixel to lie in its centre, that is, at $(x + 0.5, y + 0.5)$. (A pixel can be considered to be a rectangle of finite area with dimensions 1.0×1.0, and its sample point is the point within the pixel area where the scene is sampled in order to determine the value of the pixel.) So, for example, the intersection of an edge with a scan line is calculated for $y + 0.5$, rather than for y, as we assumed above. This is messy, and excludes the possibility of using integer-only arithmetic. A simpler solution is to assume that the sample point lies at one of the four corners of the pixel; we have chosen the top right-hand corner of the pixel. This has the consequence that the entire image is displaced half a pixel to the left and down, which in practice is insignificant. The upshot of this is that it provides the following simple rasterization rules:

(1) Horizontal edges are simply discarded.

(2) An edge which goes from scan line y_{bottom} to y_{top} should generate x values for scan lines y_{bottom} through to $y_{\text{top}}-1$ (that is missing the top scan line), or if $y_{\text{bottom}} = y_{\text{top}}$ then it generates no values.

(3) Similarly, horizontal segments should be filled from x_{left} to $x_{\text{right}}-1$ (with no pixels generated if $x_{\text{left}} = x_{\text{right}}$).

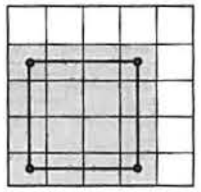


Figure 6.19
Problems with polygon boundaries – a 9-pixel polygon fills 16 pixels.

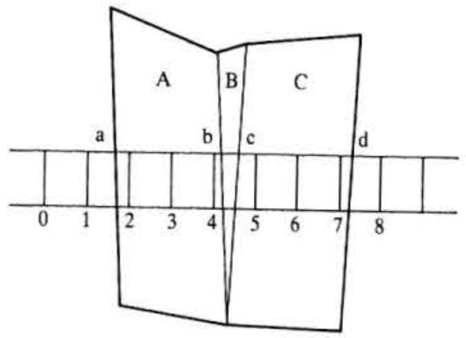


Figure 6.20
Three polygons intersecting a scan line.

Incidentally, in rules (2) and (3), whether the first or last element is ignored is arbitrary, and the choice is based around programming convenience. The four possible permutations of these two rules define the sample point as one of the four corners of the pixel. The effect of these rules can be demonstrated in Figure 6.20. Here we have three adjacent polygons A, B and C, with edges a, b, c and d. The rounded x values produced by these edges for the scan shown are 2, 4 ,4 and 7 respectively. Rule 3 then gives pixels 2 and 3 for polygon A, none for polygon B, and 4 to 6 for polygon C. Thus, overall, there are no gaps, and no overlapping. The reason why horizontal edges are discarded is because the edges adjacent to them will have already contributed the x values to make up the segment (for example, the base of the polygon in Figure 6.18; note also that, for the sake of simplicity, the scan conversion of this polygon was not done strictly in accordance with the rasterization rules mentioned above).

6.5 Order of rendering

There are two basic ways of ordering the rendering of a scene. These are: on a polygon-by-polygon basis, where each polygon is rendered in turn, in isolation from all the rest; and in scan line order, where the segments of all polygons in that scene which cross a given scan line are rendered, before moving on to the next scan line. In some textbooks, this classification has the habit of becoming hopelessly confused with the classification of hidden surface removal algorithms. In fact, the order of rendering a scene places restrictions upon which hidden surface algorithms can be used, but is of itself independent of the method employed for hidden surface removal. These are the common hidden surface removal algorithms that are compatible with the two methods:

- By polygon: Z-buffer.
- By scan line: Z-buffer; scan line Z-buffer, spanning scan line algorithm.

By-polygon rendering has a number of advantages. It is simple to implement, and it requires little data active at any one time. Because of this, it places no upper limit on scene complexity, unlike scan line rendering, which needs simultaneously to hold in memory rasterization, shading, and perhaps texture information for all polygons which cross a particular scan line. The main drawback of by-polygon rendering is that it does not make use of possible efficiency measures such as sharing information between polygons (for example, most edges in a scene are shared between two polygons). The method can only be used with the Z-buffer hidden surface algorithm, which as we shall see, is rather expensive in terms of memory usage. Also, scan-line-based algorithms possess the property that the completed image is generated in scan line order, which has advantages for hardware implementation and anti-aliasing operations.

An important difference between the two rendering methods is in the construction of the edge list. This has been described in terms of rendering on a polygon-by-polygon basis. If, however, rendering is performed in scan line order, two problems arise. One is that rasterizing all edges of all polygons in advance would consume a vast quantity of memory, setting an even lower bound on the maximum complexity of a scene. Instead, it is usual to maintain a list of 'active edges'. When a new scan line is started, all edges which start on that scan line are added to the list, whilst those which end on it are removed. For each edge in the active list, current values for x, shading information etc. are stored, along with the increments for these values. Each time a new edge is added, these values are initialized; then the increments are added for each new scan line.

The other problem is in determining segments, since there are now multiple polygons active on a given scan line. In general, some extra information will need to be stored with each edge in the active edge list, indicating which polygon it belongs to. The exact details of this depend very much upon the hidden surface removal algorithm in use. Usually an active polygon list is maintained that indicates the polygons intersected by the current scan line, and those therefore that can generate active edges. This list is updated on every scan line, new polygons being added and inactive ones deleted.

The outline of a polygon-by-polygon renderer is thus:

```
for each polygon do
  construct an edge list from the polygon edges
  for y := ymin to ymax do
    for each pair (x_i, x_i+1) in EdgeList[y] do
      shade the horizontal segment (x_i ,y) to (x_i+1,y)
```

whilst that of a scan line renderer is:

```
clear active edge list
for each scan line do
  for each edge starting on that scan line do
    add edge to active edge list
    initialize its shading and rasterization values and their increments
    remove edges which end on that scan line
    parse active edge list to obtain and render segments
    add the increments to all active edges
```

Finally, it is worth pointing out that it is possible to achieve a hybrid of these two methods. Often it is possible to split a scene up into a number of unconnected objects. If the scene is rendered on an object-by-object basis, using scan line ordering within each object, then the advantage of shared information is realized within each object, and there is no upper limit on scene complexity, only on the complexity of each individual object.

6.6 Hidden surface removal

The major hidden surface removal algorithms are described in most computer graphics textbooks and are classified in an early, but still highly relevant, paper by Sutherland *et al.* (1974) entitled 'A characterization of ten hidden-surface algorithms'. In this paper algorithms are characterized as to whether they operate primarily in object space or image (screen) space and the different uses of 'coherence' that the algorithms employ. Coherence is a term used to describe the process where geometrical units, such as areas or scan line segments, instead of single points, are operated on by the hidden surface removal algorithm.

There are two popular approaches to hidden surface removal. These are scan-line-based systems and Z-buffer-based systems. Other approaches to hidden surface removal such as area subdivision (Warnock 1969), or depth list schemes (Newell *et al.* 1972) are not particularly popular or are reserved for special-purpose applications such as flight simulation.

6.6.1 The Z-buffer algorithm

The Z-buffer algorithm, developed by Catmull (1975), is as ubiquitous in computer graphics as the Phong reflection model and interpolator, and the combination of these represents the most popular rendering option. Using Sutherland's classification scheme (Sutherland *et al.* 1974), it is an algorithm that operates in image, that is, screen space.

Pixels in the interior of a polygon are shaded, using an incremental shading scheme, and their depth is evaluated by interpolation from the z values of the polygon vertices after the viewing transformation has been applied. The equations in Section 1.5 are used to interpolate the depth values.

The Z-buffer algorithm is equivalent, for each point (x_s, y_s) to a search through the associated z values of each interior polygon point, to find that point with the minimum z value. This search is conveniently implemented by using a Z-buffer, that holds for a current point (x, y) the smallest z value so far encountered. During the processing of a polygon we either write the intensity of a point (x, y) into the frame buffer, or not, depending on whether the depth z, of the current point, is less than the depth so far encountered as recorded in the Z-buffer.

One of the major advantages of the Z-buffer is that it is independent of object representation form. Although we see it used most often in the context of polygon mesh rendering, it can be used with any representation – all that is required is the ability to calculate a z value for each point on the surface of an object. It can be used with CSG objects and separately rendered objects can be merged into a multiple object scene using Z-buffer information on each object. These aspects are examined shortly.

The overwhelming advantage of the Z-buffer algorithm is its simplicity of implementation. Its main disadvantage is the amount of memory required for the Z-buffer. The size of the Z-buffer depends on the accuracy to which the depth value of each point (x, y) is to be stored; which is a function of scene complexity. Between 20 and 32 bits is usually deemed sufficient and the scene has to be scaled to this fixed range of z so that accuracy within the scene is maximized. Recall in the previous chapter that we discussed the compression of z_s values. This means that a pair of distinct points with different z_v values can map into identical z_s values. Note that for frame buffers with less than 24 bits per pixel, say, the Z-buffer will in fact be larger than the frame buffer. In the past, Z-buffers have tended to be part of the main memory of the host processor, but now graphics terminals are available with dedicated Z-buffers and this represents the best solution.

The memory problem can be alleviated by dividing the Z-buffer into strips or partitions in screen space. The price paid for this is multiple passes through the geometric part of the renderer. Polygons are fetched from the database and rendered if their projection falls within the Z-buffer partition in screen space.

An interesting use of the Z-buffer is suggested by Foley *et al.* (1989). This involves rendering selected objects but leaving the Z-buffer contents unmodified by such objects. The idea can be applied to interaction where a three-dimensional cursor object can be moved about in a scene. The cursor is the selected object, and when it is rendered in its current position, the Z-buffer is not written to. Nevertheless the Z-buffer is used to perform hidden surface removal on the object and will move about the scene obscuring some objects and being obscured by others.

6.6.2 Z-buffer and CSG representation

The Z-buffer algorithm can be used to advantage in rendering CSG objects. As you will recall from Section 4.3, which describes a ray tracing algorithm for rendering such objects, rendering involves calculating a boundary representation of a complex object that is made up of primitive objects combined with Boolean operators and described or represented by a construction tree.

The problem with the ray tracing method is expense. A normal recursive ray tracer is a method that finds intersections between a ray of arbitrary direction and objects in the scene. This model operates recursively to any depth to evaluate specular interaction. However, with CSG objects, all rays are parallel and we are only interested in the first hit, so in this respect ray tracing is inappropriate and a Z-buffer approach is easier to implement and less expensive (Rossignac and

Requicha 1986). The Z-buffer algorithm is driven from object surfaces rather than pixel-by-pixel rays. Consider the overall structure of both algorithms.

```
● Ray tracing
  for each pixel do
    generate a ray and find all object surfaces that intersect the ray
    evaluate the CSG tree to determine the boundary of the first surface along the
    ray
    apply Z-buffer algorithm and shade or not
● Z-buffer
  for each primitive object do
    for primitive surface F do
      for each point P in a sufficiently dense grid on F do
        Project P and apply Z-buffer
        if currently visible then
          if by evaluating the CSG tree P is on boundary surface then
            render into frame buffer
```

Both algorithms have to apply a test that descends the CSG tree and evaluates the Boolean set operations to find the boundary of the object, but the Z-buffer avoids the intersection tests associated with each pixel ray.

6.6.3 Z-buffer and compositing

An important advantage of Z-buffer-based algorithms is that the z value associated with each pixel can be retained and used to enable the compositing or merging of separately generated scene elements.

Three-dimensional images are often built up from separate sub-images. A system for compositing separate elements in a scene, pixel-by-pixel, was proposed by Porter and Duff (1984) and Duff (1985). This simple system is built round an RGBαZ representation for pixels in a sub-image. The α parameter allows sub-images to be built up separately and combined, retaining sub-pixel information that may have been calculated in the rendering of each sub-image.

Composites are built up using a binary operator combining two sub-images:

$$c = f \textbf{ op } b$$

For example, consider the operator $Z_{\min}$. We may have two sub-images, say of single objects, that have been rendered separately, the Z values of each pixel in the final rendering contained in the Z channel. Compositing in this context means effecting hidden surface removal between the objects and is defined as:

$$RGB_c = (\textbf{if } Z_f < Z_b \textbf{ then } RGB_f \textbf{ else } RGB_b)$$
$$Z_c = \min(Z_f, Z_b)$$

for each pixel.

The α parameter ($0 \le \alpha \le 1$) is the fraction of the pixel area that the object covers. It is used as a factor that controls the mixing of the colours in the two images. The use of the α channel effectively extends area anti-aliasing across the compositing of images. Of course, this parameter is not normally calculated by a basic Z-buffer renderer and because of this, the method is only suitable when used in conjunction with the A-buffer hidden surface removal method (Carpenter 1984), an anti-aliased extension to the Z-buffer described in Section 14.6.

The operator **over** is defined as:

$$RGB_c = RGB_f + (1 - \alpha_f)RGB_b$$
$$\alpha_c = \alpha_f + (1 - \alpha_f)\alpha_b$$

This means that as α_f decreases more of RGB_b is present in the pixel.

The compositing operator **comp** combines both the above operators. This evaluates pixel results when Z values at the corners of pixels are different between RGB_f and RGB_b. Z_f is compared with Z_b at each of the four corners. There are 16 possible outcomes to this and if the Z values are not the same at the four corners, then the pixel is said to be confused. Linear interpolation along the edges takes place and a fraction β computed (the area of the pixel where f is in front of b). We then have the **comp** operator:

$$RGB_c = \beta(f \textbf{ over } b) + (1 - \beta)(b \textbf{ over } f)$$

Another example of compositing in three-dimensional image synthesis is given in Nakamae *et al.* (1986). This is a montage method, the point of which is to integrate a three-dimensional computer generated image, of say a new building, with a real photograph of a scene. The success of the method is due to the fact that illuminance for the generated object, is calculated from measurements of the background photograph and because atmospheric effects are integrated into the finished image.

With the decreasing cost of memory and the popularity of the OpenGL, the use of accumulation buffers has become common. A powerful and simple use of accumulation buffers is in their support for multi-pass rendering techniques. This approach is described in Section 6.7.

6.6.4 Z-buffer and rendering

The Z-buffer imposes no constraints on database organization (other than those required by the shading interpolation) and in its simplest form can be driven on a polygon-by-polygon basis, with polygons being presented in any convenient order.

In principle, for each polygon we compute:

(1) The (x, y) value of the interior pixels.

(2) The z depth for each point (x, y).

(3) The intensity, I, for each point (x, y).

Thus we have three concurrent bilinear interpolation processes and a triple nested loop. The z values and intensities, I, are available at each vertex and the interpolation scheme for z and I is distributed between the two inner loops of the algorithm.

An extended version of the by-polygon algorithm with Z-buffer hidden surface removal is as follows:

```
for all x, y do
  Z-Buffer[x, y] := maximum_depth
for each polygon do
  construct an edge list from the polygon edges (that is, for each edge, calculate the
  values of x, z and I for each scan line by interpolation and store them in the edge
  list)
for y := $y_{min}$ to $y_{max}$ do
  for each segment in EdgeList[y] do
    get $X_{left}, X_{right}, Z_{left}, Z_{right}, I_{left}, I_{right}$
      for x := $X_{left}$ to $X_{right}$ do
      linearly interpolate z and I between $Z_{left}, Z_{right}$ and $I_{left}, I_{right}$ respectively
      if z < Z_Buffer[x,y] then
        Z_Buffer[x,y] := z
        frame_buffer[x,y] := I
```

The structure of the algorithm reveals the major inefficiency of the method in that shading calculations are performed on hidden pixels which are then either ignored or subsequently overwritten.

If Phong interpolation is used then the final reflection model calculations, which are a function of the interpolated normal, should also appear within the innermost loop; that is, interpolate $\mathbf{N}$ rather than I, and replace the last line with:

frame_buffer[x,y] := *ShadingFunction*($\mathbf{N}$)

6.6.5 Scan line Z-buffer

There is a variation of the Z-buffer algorithm for use with scan-line-based renderers, known (not suprisingly) as a scan line Z-buffer. This is simply a Z-buffer which is only one pixel high, and is used to solve the hidden surface problem for a given scan line. It is re-initialized for each new scan line. Its chief advantage lies in the small amount of memory it requires relative to a full-blown Z-buffer; and it is common to see a scan line Z-buffer-based program running on systems which do not have sufficient memory to support a full Z-buffer.

6.6.6 Spanning hidden surface removal

A spanning hidden surface removal algorithm attempts, for each scan line, to find 'spans' across which shading can be performed. The hidden surface removal problem is thus solved by dividing the scan line into lengths over which a single surface is dominant. This means that shading calculations are performed only once per pixel, removing the basic inefficiency inherent in the Z-buffer method. Set against this is the problem that spans do not necessarily correspond to polygon segments, making it harder to perform incremental shading calculations (the start values must be calculated at an arbitary point along a polygon segment, rather than being set to the values at the left-hand edge). The other major drawback is in the increase in complexity of the algorithm itself, as will be seen.

It is generally claimed that spanning algorithms are more efficient than Z-buffer-based algorithms, except for very large numbers of polygons (Foley *et al.* 1989; Sutherland *et al.* 1974). However, since extremely complex scenes are now becoming the norm, it is becoming clear that overall, the Z-buffer is more efficient, unless a very complex shading function is being used.

6.6.7 A spanning scan line algorithm

The basic idea, as has been mentioned, is that rather than solving the hidden surface problem on a pixel-by-pixel basis using incremental z calculation, the spanning scan line algorithm uses spans along the scan line over which there is no depth conflict. The hidden surface removal process uses coherence in x and deals in units of many pixels. The processing implication is that a sort in x is required for each scan line and the spans have to be evaluated.

The easiest way to see how a scan line algorithm works is to consider the situation in three-dimensional screen space (x_s, y_s, z_s). A scan line algorithm effectively moves a scan line plane, that is a plane parallel to the (x_s, z_s) plane, down the y_s axis. This plane intersects objects in the scene and reduces the hidden surface problem to two-dimensional space (x_s, z_s). Here the intersection of the scan line plane with object polygons become lines (Figure 6.21). These line segments are then compared to solve the hidden surface problem by considering 'spans'. A span is that part of a line segment that is contained between the edge intersections of all active polygons. A span can be considered a coherence unit, within the extent of which the hidden surface removal problem is 'constant' and can be solved by depth comparisons at either end of the span. Note that a more complicated approach has to be taken if penetrating polygons are allowed.

It can be seen from this geometric overview that the first stage in a spanning scan line algorithm is to sort the polygon edges by y_s vertex values. This results in an active edge list which is updated as the scan line moves down the y_s axis. If penetrating polygons are not allowed, then each edge intersection with the current scan line specifies a point on the scan line where 'something is changing', and so these collectively define all the span boundary points.

By going through the active edge list in order, it is possible to generate a set of line segments, each of which represents the intersection of the scan line plane with a polygon. These are then sorted in order of increasing x_s.

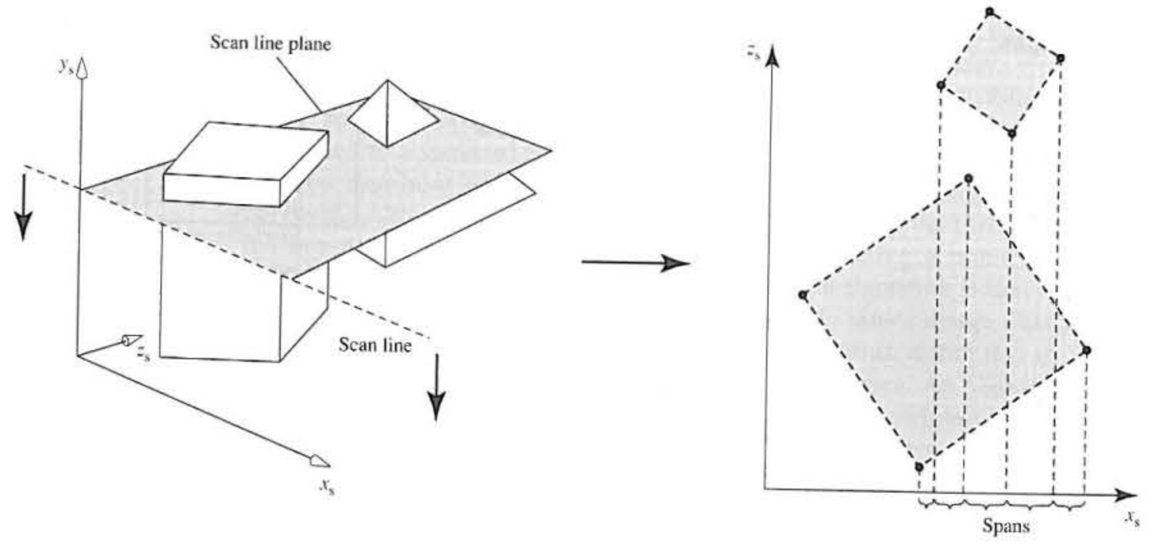


Figure 6.21
A scan line plane is moved down through the scene producing line segments and spans.

The innermost loop then processes each span in the current scan line. Active line segments are clipped against span boundaries and are thus subdivided by these boundaries. The depth of each subdivided segment is then evaluated at one of the span boundaries and hidden surface removal is effected by searching within a span for the closest subdivided segment. This process is shown in Figure 6.22.

In pseudo-code the algorithm is:

```
for each polygon do
  Generate and bucket sort in $y_s$ the polygon edge information
for each scan line do
  for each active polygon do
    Determine the segment or intersection of the scan plane and polygon
    Sort these active segments in $x_s$
    Update the rate of change per scan line of the shading parameters
    Generate the span boundaries
      for each span do
      Clip active segments to span boundaries
      Evaluate the depth for all clipped segments at one of the span boundaries
      Solve the hidden surface problem for the span with minimum $z_s$
      Shade the visible clipped line segment
      Update the shading parameters for all other line segments by the rate of
      change per pixel times the span width
```

Note that integrating shading information is far more cumbersome than with the Z-buffer. Records of values at the end of clipped line segments have to be kept and updated. This places another scene complexity overhead (along with the absolute number of polygons overhead) on the efficiency and memory requirements of the process.

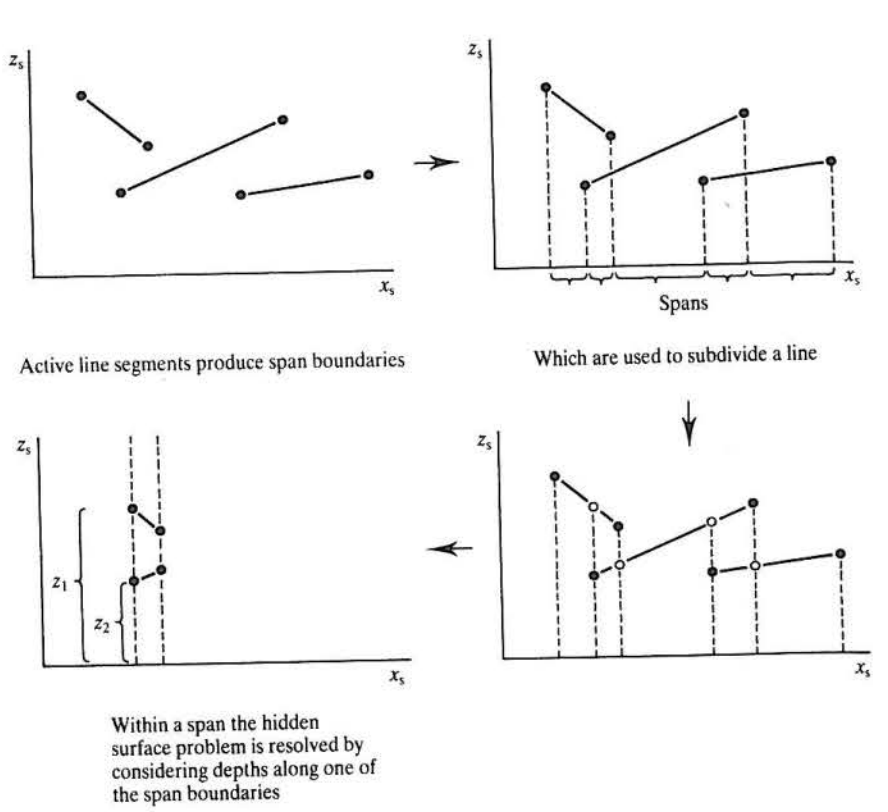

z_s
x_s
Active line segments produce span boundaries
Spans
Which are used to subdivide a line
z_1
z_2
Within a span the hidden surface problem is resolved by considering depths along one of the span boundaries

Figure 6.22
Processing spans.

6.6.8 Z-buffer and complex scenes

Research into hidden surface removal tended to predominate in computer graphics in the 1970s but then with the industry acceptance of the Z-buffer algorithm, hidden surface removal was regarded as solved and the main research effort in rendering moved towards light transport models in the 1980s. (Although research continued into methods of improving the Z-buffer algorithm – dealing with aliasing and image composition methods.) The Z-buffer algorithm suffers from well-known efficiency problems and perhaps the demands of virtual reality image generation will re-emphasize the importance of efficient hidden surface removal.

The disadvantage of the Z-buffer is that it causes unseen polygons to be rendered. If we wish to retain the advantages of the traditional Z-buffer approach then this is a major cost factor that we must avoid in complex scenes. The main advantage of the Z-buffer, apart from its simplicity, is that the computing cost per polygon is low. It exploits image space coherence and adjacent pixels in a polygon projection are handled by a simple incremental calculation. However, this advantage rapidly diminishes as the scene to be rendered becomes more and more complex and the polygons smaller and smaller. The set-up computations at the polygon vertices predominate over the pixel-by-pixel calculations.

Depth complexity of a scene is a function of the number of objects in a scene just as object complexity is, and scenes that we wish to render in virtual reality applications will tend to possess both depth and object complexity.

An approach, reported by Greene *et al.* in 1993 takes the basic Z-buffer algorithm and develops it into a method that is suitable for complex scenes. Compared to a standard Z-buffer, Greene reports a reduction in rendering time from over an hour to 6.45 seconds on a complex scene containing 53 000 000 polygons. (Although an important practical factor that enables this reduction is the fact that the large scene is constructed by replicating a smaller scene of 15 000 polygons.) Another important attribute of the algorithm is that it can be implemented on existing hardware designs with only minor design changes.

The reason for the inefficiency of the Z-buffer algorithm is that it is an image space algorithm. It cannot exploit object space coherence. An algorithm that does exploit object space coherence is the hidden surface element in a ray tracing algorithm that uses a spatial subdivision scheme (an octree, for example) for ray tracking. For each ray cast, the first surface that the ray hits is the visible surface and no surfaces behind the one intersected are considered by the algorithm. (A ray tracing algorithm, on the other hand, does not exploit image space coherence and each pixel calculation is independent of every other.) Greene's development recognizes this and employs spatial subdivision to incorporate object space coherence in the traditional Z-buffer. It also uses a so-called Z-pyramid to further accelerate the traditional Z-buffer image space coherence.

The object space coherence is set up by constructing a conventional octree for the scene and using this to guide the rendering strategy. A node of an octree is hidden if all the faces of the cube associated with that node are hidden with respect to the Z-buffer. If such is the case then, of course, all the polygons or complete objects that the cube contains are hidden. This fact leads to the obvious rendering strategy of starting at the root of the tree and 'rendering' octree cubes, by rendering each face of the cube, to determine whether the whole cube is hidden or not. If it is not hidden we proceed with the geometry inside the cube. Thus a large number of hidden polygons are culled at the cost of rendering cube faces. So we are imposing a rendering order on the normally arbitrary polygon ordering in the Z-buffer. Greene points out that this in itself can be expensive – if the cube being rendered projects onto a large number of pixels – and this consideration leads to further exploiting the normal Z-buffer advantage of image space coherence.

The Z-pyramid is a strategy that attempts to determine complete polygon visibility without pixel-by-pixel elaboration. A Z-pyramid is a detail hierarchy with the original Z-buffer at the lowest level. At each level there is a half resolution Z-buffer, where the *z* value for a cell is obtained from the largest of the *z* values of the four cells in the next level down (you might say a depth mip-map). Maintaining the Z-pyramid involves tracking up the hierarchy in the direction of finer resolution until we encounter a depth that is already as far away as the current depth value. Using the Z-pyramid to test the visibility of a cube face involves finding the finest detail level whose corresponding projection in screen space just covers the projection of the face. Then it is simply a matter of comparing the nearest vertex depth of the face against the value in the Z-pyramid. Using the Z-pyramid to test a complete polygon for visibility is the same except that the screen space bounding box of the polygon is used.

In this way the technique tries to make the best of both object and image space coherence. Using spatial subdivision to accelerate hidden surface removal is a old idea and seems to have been first mooted by Schumaker *et al.* (1969). Here the application was flight simulation where the real time constraint was, in 1969, a formidable problem.

Temporal coherence is exploited by retaining the visible cubes from the previous frame. For the current frame the polygons within these cubes are rendered first and the cubes marked as such. The algorithm then proceeds as normal. This strategy plays on the usual event that most of the cubes from the previous frame will still be visible; a few will become invisible in the current frame and a few cubes, invisible in the previous frame, will become visible.

6.6.9 Z-buffer summary

From an ease of implementation point-of-view the Z-buffer is the best algorithm. It has significant memory requirements particularly for high resolution frame buffers. However, it places no upward limit on the complexity of scenes, an advantage that is now becoming increasingly important. It renders scenes one object at a time and for each object one polygon at a time. This is both a natural and convenient order as far as database considerations are concerned.

An important restriction it places on the type of object that can be rendered by a Z-buffer is that it cannot deal with transparent objects without costly modification. A partially transparent polygon may:

(1) Be completely covered by an opaque nearer polygon, in which case there is no problem; or,

(2) Be the nearest polygon, in which case a list of all polygons behind it must be maintained so that an appropriate combination of the transparent polygon and the next nearest can be computed. (The next nearest polygon is not, of course, known until all polygons are processed.)

Compared with scan line algorithms, anti-aliasing solutions, particularly hardware implementations, are more difficult.

Cook, Carpenter and Catmull (1987) point out that a Z-buffer has an extremely important 'system' advantage. It provides a 'back door' in that it can combine point samples with point samples from other algorithms that have other capabilities such as radiosity or ray tracing.

If memory requirements are too prodigious then a scan line Z-buffer is the next best solution. Unless a renderer is to work efficiently on simple scenes, it is doubtful if it is worth contemplating the large increase in complexity that a spanning scan line algorithm demands.

Historically there has been a shift in research away from hidden surface problems to realistic image synthesis. This has been motivated by the easy availability of high spatial and colour resolution terminals. All of the 'classic' hidden surface removal algorithms were developed prior to the advent of shading complexities and it looks as if the Z-buffer will be the most popular survivor for conventional rendering.

6.6.10 BSP trees and hidden surface removal

Having introduced the idea of BSP trees in Chapter 2, we now return to examine them in more detail; in particular how they can be used to perform hidden surface calculations.

For almost two decades after the introduction of BSP trees into computer graphics applications, their usage seems to have been restricted in the main to flight simulators. With the advent of three-dimensional video games and other animation applications on standard PCs we have seen a renewal of interest in using BSP trees for visibility calculations and we will now look at this method in some detail.

The original BSP hidden surface removal idea depends on having a static scene and a changing view point – the classic flight simulator/computer game application – and works as a two-phase process. In the first phase a BSP tree of the scene is constructed (once only, which in practice would be an off-line process) and in the second phase the view point is compared with this structure to determine visibility. Its attraction for real time graphics is that much visibility processing can take place as a pre-process. We will look first of all at the simpler issue of determining visibility amongst objects then see how the principles can be extended to determine polygon visibility within an object.

If our scene consists of convex objects that can be separated by convex regions made up of planes then we can use a recursive divide and conquer strategy to divide up the space. We assume that we have an appropriate strategy for positioning the planes and that the tree is complete when all regions only contain a single object. Figure 6.23 shows the idea. Each leaf is a label identifying an object and each node is a separating plane. Having constructed a tree (Figure 6.23(a)) we can then determine a visibility ordering for a view point descending the tree from the root node with the view point coordinates to give the object closest to the view point (Figure 6.23(b)). Starting at the root node we descend the subtree on the side of plane A nearest to the view point (in this case the negative side) taking us to the node associated with plane B and thenceforth to object 2. Figure 6.23 (c) indicates the route we take to determine a visibility ordering. Object 3 is the next nearest to the view point and returning to the root node and descending, and the remaining ordering is object 1 followed by object 4. This gives us a near to far visibility ordering which for this scene is objects 2, 3, 1 and 4. Alternatively we can just as easily generate a far to near ordering.

In practice this scheme is not particularly useful because most computer graphics applications are made up of scenes where the object complexity

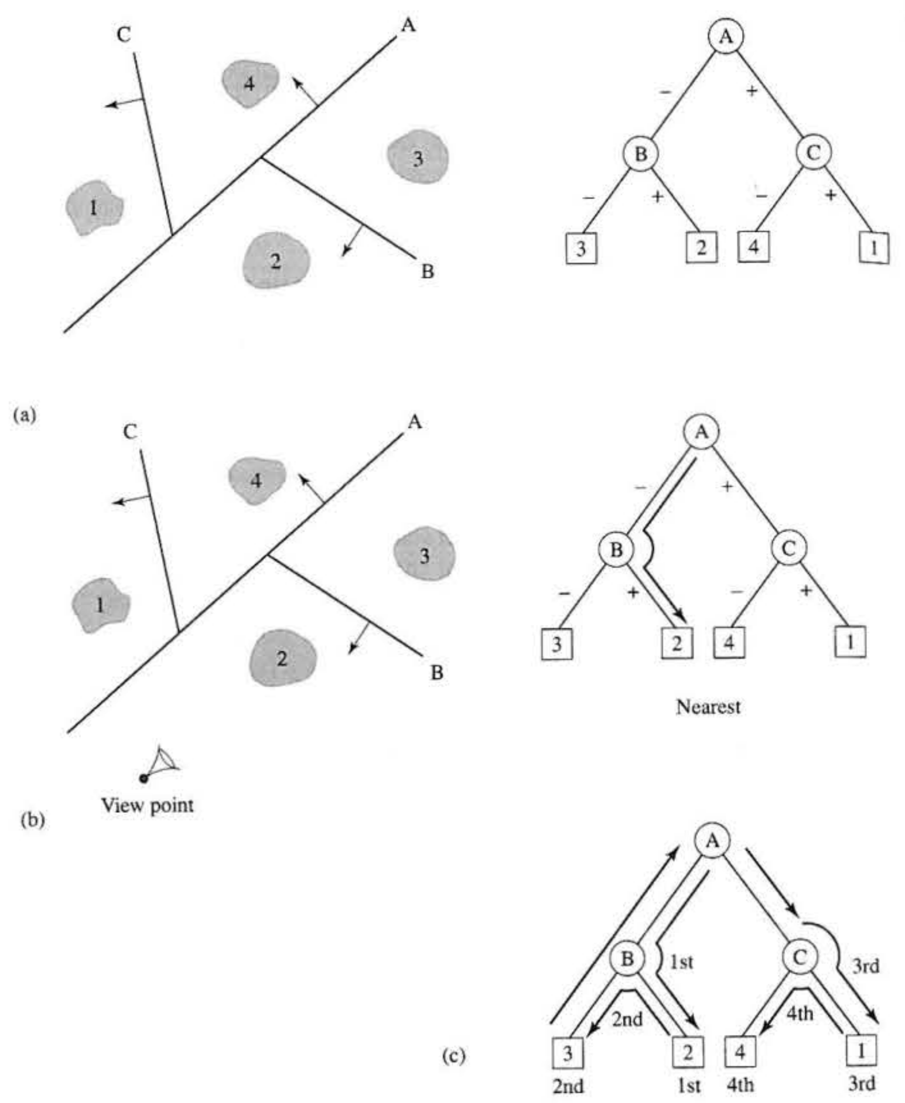


Figure 6.23
BSP operations for a four object scene.
(a) Constructing a BSP tree.
(b) Descending the tree with the view point coordinates gives the nearest object.
(c) Evaluating a visibility order for all objects.

(number of polygons per object) is much greater than scene complexity (number of objects per scene) and for the approach to be useful we have to deal with polygons within objects rather than entire objects. Also there is the problem of positioning the planes – itself a non-trivial problem. If the number of objects is small then we can have a separating plane for every pair of objects – a total of n^2 for an n object scene.

For polygon visibility ordering we can choose planes that contain the face polygons. A polygon is selected and used as a root node. All other polygons are tested against the plane containing this polygon and placed on the appropriate descendant branch. Any polygon which crosses the plane of the root polygon is

split into two constituents. The process continues recursively until all polygons are contained by a plane. Obviously the procedure creates more polygons than were originally in the scene but practice has shown that this is usually less than a factor of two.

The process is shown for a simple example in Figure 6.24. The first plane chosen, plane A, containing a polygon from object 1, splits object 3 into two parts. The tree builds up as before and we now use the convention IN/OUT to say which side of a partition an entity lies since this now has meaning with respect to the polygonal objects.

Far to near ordering was the original scheme used with BSP trees. Rendering polygons into the frame buffer in this order results in the so-called painter's algorithm – near polygons are written 'on top of' farther ones. Near to far ordering can also be used but in this case we have to mark in some way the fact that a pixel has already been visited. Near to far ordering can be advantageous in extremely complicated scenes if some strategy is adopted to avoid rendering completely occluded surfaces, for example, by comparing their image plane extents with the (already rendered) projections of nearer surfaces.

Thus to generate a visibility order for a scene we:

- Descend the tree with view point coordinates.
- At each node, we determine whether the view point is in front of or behind the node plane.
- Descend the far side subtree first and output polygons.
- Descend the near side subtree and output polygons.

This results in a back to front ordering for the polygons with respect to the current view point position and these are rendered into the frame buffer in this order. If this procedure is used then the algorithm suffers from the same efficiency disadvantage as the Z-buffer – rendered polygons may be subsequently obscured. However, one of the disadvantages of the Z-buffer is immediately overcome. Polygon ordering allows the unlimited use of transparency with no additional effort. Transparent polygons are simply composited according to their transparency value.

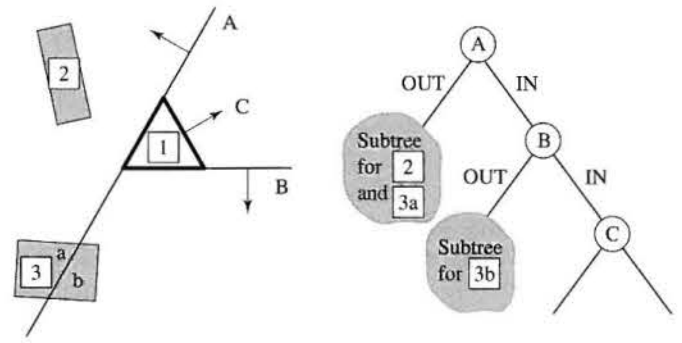


Figure 6.24
A BSP tree for polygons.

6.7 Multi-pass rendering and accumulation buffers

The rendering strategies that we have outlined have all been single pass approaches where a rendered image is composed by one pass through a graphics pipeline. In Section 6.6.3 we looked at a facility that enabled certain operations on separately rendered images with αZ components to be combined. In this section we will look at multi-pass rendering which means composing a single image of a scene from a combination of images of that scene rendered by passing it through the pipeline with different values for the rendering parameters. This approach is possible due to the continuing expansion of hardware and memory dedicated to rendering, manifested in texture mapping hardware which has substantially increased the visual complexity of real time imagery generated on a PC, and the availability of multiple screen resolution buffers such as a stencil buffer and an accumulation buffer (as well as the frame buffer and Z-buffer). The accumulation buffer is a simplified version of the A-buffer and the availability of such a facility has led to an expansion of the algorithms that employ a multi-pass technique.

As the name implies, an accumulation buffer accumulates rendered images and the standard operations are addition and multiplication combined into an 'add with weight' operation. In practice an accumulation buffer may have higher precision than a screen buffer to diminish the effect of rounding errors. The use of an accumulator buffer enables the effect of particular single pass algorithms to be obtained by a number of passes. After the passes are complete the final result in the accumulation buffer is transferred into the screen buffer.

The easiest example is the common anti-aliasing algorithm (see Section 14.7 for full details of this approach) which is to generate a virtual image, at $n \times$ the resolution of the final image, then reduce this to the final image by using a filter. The same effect can be obtained by jittering the view port and generating n images and accumulating these with the appropriate weighting value which is a function of the jitter value. In Figure 6.25, to generate the four images that are required to sample each pixel four times we displace the view window through a $^1/_2$ pixel distance horizontally and vertically. To find this displacement we only have to calculate the size of the view port in pixel units. (Note that this cannot be implemented using the simple viewing system given in Chapter 5, which assumes that the view window is always centred on the line through the view point.)

In this case we only save on memory. However, in many instances an algorithm implemented as a multi-pass rendering is of lower complexity than the single pass equivalent. Additional examples of motion blur, soft shadows and depth of field are given in Haeberli and Akeley (1990). These effects can be achieved by distributed ray tracing as described in Chapter 10 and the marked difference between the complexity of the two approaches is obvious.

To create a motion blurred image it is only necessary to accumulate a series of images rendered while the moving objects in the scene change their position over

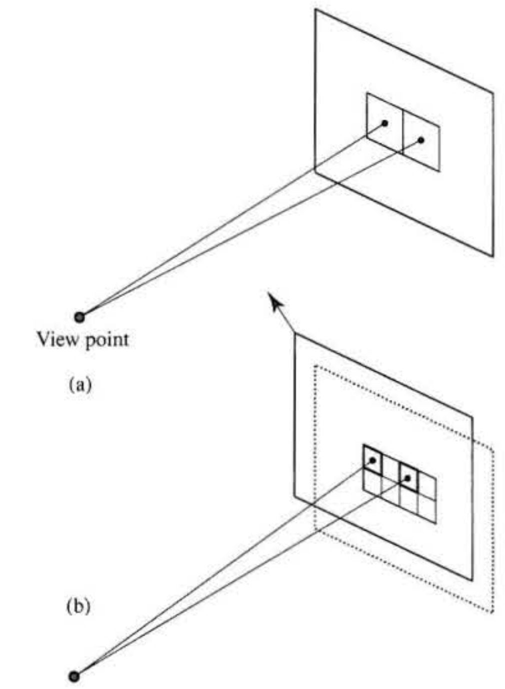


Figure 6.25
Multi-pass super-sampling. (a) Aliased image (1 sample/pixel). (b) A one component/pass of the anti-aliased image (four samples/pixel or four passes). For this pass the view point is moved up and to the left by $^1/_2$ pixel dimension).

time. Exactly analogous to the anti-aliasing example, we are now anti-aliasing in the time domain. There are two approaches to motion blur. We can display a single image by averaging n images built up in the accumulation buffer. Alternatively we can display an image for every calculated image by averaging over a window of n frames moving in time. To do this we accumulate n images initially. At the next step the frame that was accumulated n–1 frames ago is re-rendered and subtracted from the accumulation buffer. Then the contents of the accumulation buffer are displayed. Thus, after the initial sequence is generated, each time a frame is displayed two frames have to be rendered – the $(n-1)$th and the current one.

Simulating depth of field is achieved (approximately) by jittering both the view window as was done for anti-aliasing and the view point. Depth of field is the effect seen in a photograph where, depending on the lens and aperture setting, objects a certain distance from the camera are in focus where others nearer and farther away are out of focus and blurred. Jittering the view window makes all objects out of focus and jittering the view point at the same time ensures objects in the equivalent of the focal plane remain in focus. The idea is shown in Figure 6.26. A plane of perfect focus is decided on. View port jitter values and view point perturbations are chosen so that a common rectangle is maintained in the plane of perfect focus. The overall transformation applied to the view frustum is a shear and translation. Again this facility cannot be implemented using the simple view frustum in Section 5.2 which does not admit shear projections.

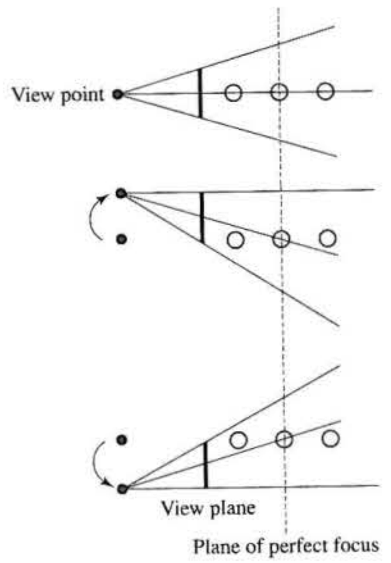


Figure 6.26
Simulating depth of field by shearing the view frustum and translating the view point.

Soft shadows are easily created by accumulating n passes and changing the position of a point light source between passes to simulate sampling of an area source. Clearly this approach will also enable shadows from separate light sources to be rendered.

7

Simulating light–object interaction: local reflection models

Introduction

Local reflection models, and in particular the Phong model (introduced in Chapter 5), have been part of mainstream rendering since the mid-1970s. Combined with interpolative shading of polygons, local reflection models are incorporated in almost every conventional renderer. The obvious constraint of locality is the strongest disadvantage of such models but despite the availability of ray tracers and radiosity renderers the mainstream rendering approach is still some variation of the strategy described earlier – in other words a local reflection model is at the heart of the process. However, nowadays it would be difficult to find a renderer that did not have ad hoc additions such as texture mapping and shadow calculation (see Chapters 8 and 9). Texture mapping adds interest and variety, and geometrical shadow calculations overcome the most significant drawback of local models.

Despite the understandable emphasis on the development of global models, there has been some considerable research effort into improving local reflection models. However, not too much attention has been paid to these, and most

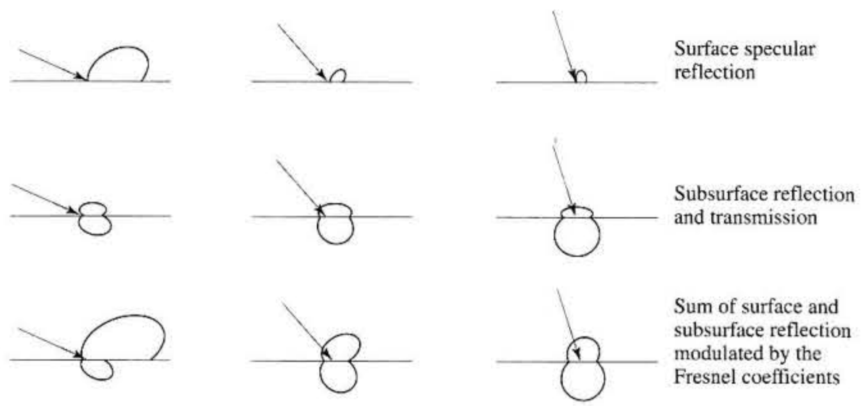

Surface specular reflection
Subsurface reflection and transmission
Sum of surface and subsurface reflection modulated by the Fresnel coefficients

Figure 7.10 Reflection behaviour due to Hanrahan and Kreuger's model (after Hanrahan and Kreuger (1993)).

of incidence of the light. For a plane surface the amount of light entering the surface depends on Fresnel's law – the more light that enters the surface, the higher will be the contribution or influence from subsurface events to the total reflected light L_r. So the influence of L_{rv} depends on the angle of incidence. Subsurface scattering depends on the physical properties of the material. A material is modelled by a suspension of scattering sites or particles and parametrized by absorption and scattering cross-sections. These express the probability of occurrence per unit path length of scattering or absorption. The relative size of these parameters determines whether the scattering is forward, backward or isotropic.

The effect of these two factors is shown, for a simple case, in Figure 7.10. The first row shows high/low specular reflection as a function of angle of incidence. The behaviour of reflected light is dominated by surface scattering or specular reflection when the angle of incidence is high and by subsurface scattering when the angle of incidence is low. As we have seen, this behaviour is modelled, to a certain extent, by the Cook and Torrance model of Section 7.6. The second row shows reflection lobes due to subsurface scattering and it can be seen that materials can exhibit backward, isotropic or forward scattering behaviour. (The bottom lobes do not, of course, contribute to L_r but are nevertheless important when considering materials that are made up of multiple layers and thin translucent materials that are backlit.) The third row shows that the combination of L_{rs} and L_{rv} will generally result in non-isotropic behaviour which exhibits the following general attributes:

- Reflection increases as material layer thickness increases due to increased subsurface scattering.
- Subsurface scattering can be backward, isotropic or forward.
- Reflection from subsurface scattering tends to produce functions that are flattened on top of the lobe compared with the (idealized) hemisphere of Lambert's law.

Such factors result in the subtle differences between the model and Lambert's law.

8 Mapping techniques

Introduction

In this chapter we will look at techniques which store information (usually) in a two-dimensional domain which is used during rendering to simulate textures. The mainstream application is texture mapping but many other applications are described such as reflection mapping to simulate ray tracing. With the advent of texture mapping hardware the use of such facilities to implement real time rendering has seen the development of light maps. These use the texture facilities to enable the pre-calculation of (view independent) lighting which then 'reduces' rendering to a texture mapping operation.

Texture mapping became a highly developed tool in the 1980s and was the technique used to enhance Phong shaded scenes so that they were more visually interesting, looked more realistic or esoteric. Objects that are rendered using only Phong shading look plastic-like and texture mapping is the obvious way to add interest without much expense.

Texture mapping developed in parallel with research into global illumination algorithms – ray tracing and radiosity (see Chapters 10, 11 and 12). It was a device that could be used to enhance the visual interest of a scene, rather than its photo-realism and its main attraction was cheapness – it could be grafted onto a standard rendering method without adding too much to the processing cost. This contrasted to the global illumination methods which used completely different algorithms and were much more expensive than direct reflection models.

Another use of texture mapping that became ubiquitous in the 1980s was to add pseudo-realism to shiny animated objects by causing their surrounding environment to be reflected in them. Thus tumbling logos and titles became chromium and the texture reflected on them moved as the objects moved. This technique – known as environment mapping – can also be used with a real photographed environment and can help to merge a computer animated object with a real environment. Environment mapping does not accomplish anything that could not be achieved by ray tracing – but it is much more efficient. A more recent use of environment mapping techniques is in image-based rendering which is discussed in Chapter 16.

As used in computer graphics, 'texture' is a somewhat confusing term and generally does not mean controlling the small-scale geometry of the surface of a computer graphics object – the normal meaning of the word. It is easy to modulate the colour of a Phong shaded object by controlling the value of the three diffuse coefficients and this became the most common object parameter to be controlled by texture mapping. (Colour variations in the physical world are not, of course, generally regarded as texture.) Thus as the rendering proceeds at pixel-by-pixel level, we pick up values for the Phong diffuse reflection coefficients and the diffuse component (the colour) of the shading changes as a function of the texture map(s). A better term is colour mapping and this appears to be coming into common usage.

This simple pixel-level operation conceals many difficulties and the geometry of texture mapping is not straightforward. As usual we make simplifications that lead to a visually acceptable solution. There are three origins to the difficulties:

(1) We mostly want to use texture mapping with the most popular representation in computer graphics – the polygon mesh representation. This, as we know, is a geometric representation where the object surface is approximated, and this approximation is only defined at the vertices. In a sense we have no surface – only an approximation to one – so how can we physically derive a texture value at a surface point if the surface does not exist?

(2) We want to use, in the main, two-dimensional texture maps because we have an almost endless source of textures that we can derive by frame-grabbing the real world, by using two-dimensional paint software or by generating textures procedurally. Thus the mainstream demand is to map a two-dimensional texture onto a surface that is approximated by a polygon mesh. This situation has become consolidated with the advent of cheap texture mapping hardware facilities.

(3) Aliasing problems in texture mapping are usually highly visible. By definition textures usually manifest some kind of coherence or periodicity. Aliasing breaks this up and the resulting mess is usually high visible. This effect occurs as the periodicity in the texture approaches the pixel resolution.

We now list the possible ways in which certain properties of a computer graphics model can be modulated with variations under control of a texture map. We have listed these in approximate order of their popularity (which also tends to relate to their ease of use or implementation). These are:

(1) **Colour** As we have already pointed out, this is by far the most common object property that is controlled by a texture map. We simply modulate the diffuse reflection coefficients in the Phong reflection model with the corresponding colour from the texture map. (We could also change the specular coefficients across the surface of an object so that it appears shiny and matte as a function of the texture map. But this is less common, as being able to perceive this effect on the rendered object depends on producing specular highlights on the shiny parts if we are using the basic Phong reflection model.)

(2) **Specular 'colour'** This technique – known as environment mapping or chrome mapping – is a special case of ray tracing where we use texture map techniques to avoid the expense of full ray tracing. The map is designed so that it looks as if the (specular) object is reflecting the environment or background in which it is placed.

(3) **Normal vector perturbation** This elegant technique applies a perturbation to the surface normal according to the corresponding value in the map. The technique is known as bump mapping and was developed by a famous pioneer of three-dimensional computer graphic techniques – J. Blinn. The device works because the intensity that is returned by a Phong shading equation reduces, if the appropriate simplifications are made, to a function of the surface normal at the point currently being shaded. If the surface normal is perturbed then the shading changes and the surface that is rendered looks as if it is textured. We can therefore use a global or general definition for the texture of a surface which is represented in the database as a polygon mesh structure.

(4) **Displacement mapping** Related to the previous technique, this mapping method uses a height field to perturb a surface point along the direction of its surface normal. It is not a convenient technique to implement since the map must perturb the geometry of the model rather than modulate parameters in the shading equation.

(5) **Transparency** A map is used to control the opacity of a transparent object. A good example is etched glass where a shiny surface is roughened (to cause opacity) with some decorative pattern.

There are many ways to perform texture mapping. The choice of a particular method depends mainly on time constraints and the quality of the image required. To start with we will restrict the discussion to two-dimensional texture maps – the most popular and common form – used in conjunction with polygon mesh objects. (Many of the insights detailed in this section are based on descriptions in Heckbert's (1986) defining work in this area.)

Mapping a two-dimensional texture map onto the surface of an object then projecting the object into screen space is a two-dimensional to two-dimensional transformation and can thus be viewed as an image warping operation. The most common way to do this is to inverse map – for each pixel we find its corresponding 'pre-image' in texture space (Figure 8.1(b)). However, for reasons that will shortly become clear, specifying this overall transformation is not straightforward and we consider initially that texture mapping is a two-stage process that takes us from the two-dimensional texture space into the three-dimensional space of the object and then via the projective transform into two-dimensional screen space (Figure 8.1(a)). The first transformation is known as parametrization and the second stage is the normal computer graphics projective transformation. The parametrization associates all points in texture space with points on the object surface.

The use of an anti-aliasing method is mandatory with texture mapping. This is easily seen by considering an object retreating away from a viewer so that its projection in screen space covers fewer and fewer pixels. As the object size decreases, the pre-image of a pixel in texture space will increase covering a larger area. If we simply point sample at the centre of the pixel and take the value of $T(u, v)$ at the corresponding point in texture space, then grossly incorrect results will follow (Figure 8.2(a), (b) and (c)). An example of this effect is shown in Figure 8.3. Here, as the chequerboard pattern recedes into the distance, it begins to break up in a disturbing manner. These problems are highly visible and move when animated. Consider Figure 8.2(b) and (c). Say, for example, that an object projects onto a single pixel and moves in such a way that the pre-image translates across the $T(u, v)$. As the object moves it would switch colour from black to white.

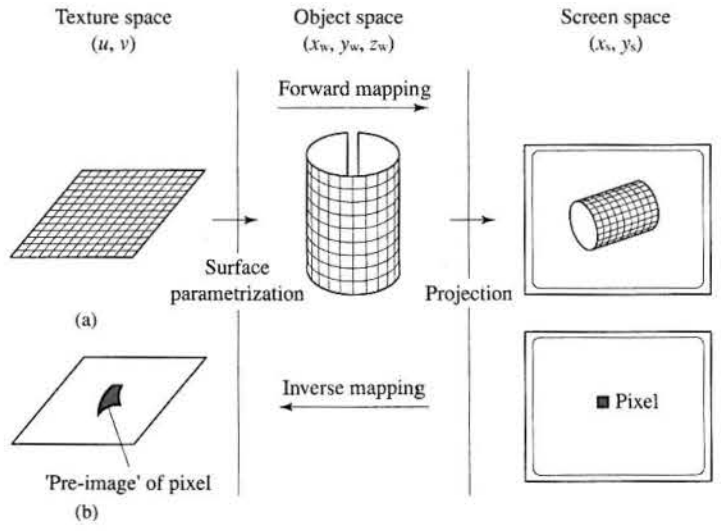
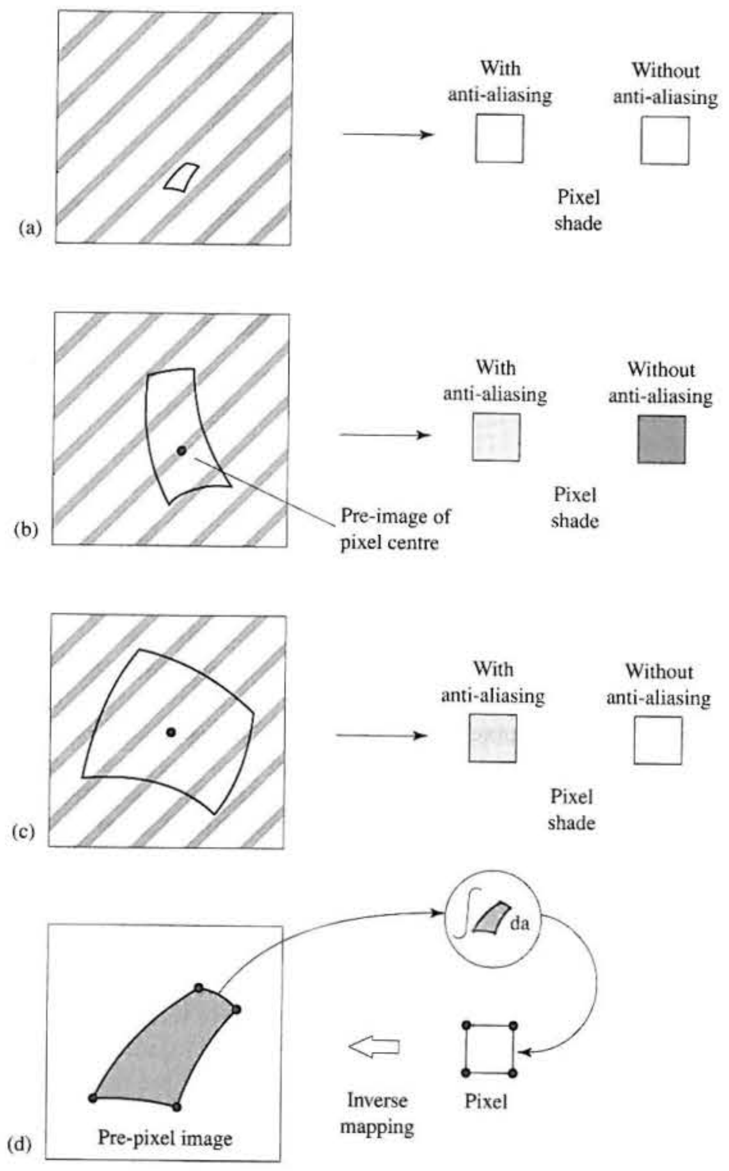


Figure 8.1
Two ways of viewing the process of two-dimensional texture mapping.
(a) Forward mapping.
(b) Inverse mapping.

Figure 8.2
Pixels and pre-images in $T(u,v)$ space.



Anti-aliasing in this context then means integrating the information over the pixel pre-image and using this value in the shading calculation for the current pixel (Figure 8.2(d)). At best we can only approximate this integral because we have no knowledge of the shape of the quadrilateral, only its four corner points.

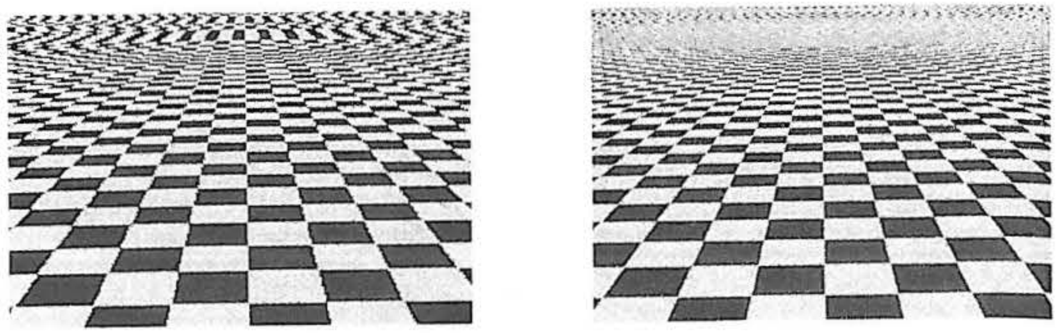


Figure 8.3
Aliasing in texture mapping. The pattern in (b) is a super-sampled (anti-aliased) version of that in (a). Aliases still occur but appear at a higher spatial frequency.

8.1 Two-dimensional texture maps to polygon mesh objects

The most popular practical strategy for texture mapping is to associate, during the modelling phase, texture space coordinates (u, v) with polygon vertices. The task of the rendering engine then is to find the appropriate (u, v) coordinate for pixels internal to each polygon. The main problem comes about because the geometry of a polygon mesh is only defined at the vertices – in other words there is no analytical parametrization possible. (If the object has an analytical definition – a cylinder, for example – then we have a parametrization and the mapping of the texture onto the object surface is trivial.)

There are two main algorithm structures possible in texture mapping, inverse mapping (the more common) and forward mapping. (Heckbert refers to these as screen order and texture order algorithms respectively.) Inverse mapping (Figure 8.1(b)) is where the algorithm is driven from screen space and for every pixel we find by an inverse mapping its 'pre-image' in texture space. For each pixel we find its corresponding (u, v) coordinates. A filtering operation integrates the information contained in the pre-image and assigns the resulting colour to the pixel. This algorithm is advantageous if the texture mapping is to be incorporated into a Z-buffer algorithm where the polygon is rasterized and depth and lighting interpolated on a scan line basis. The square pixel produces a curvilinear quadrilateral as a pre-image.

In forward mapping the algorithm is driven from texture space. This time a square texel in texture space produces a curvilinear quadrilateral in screen space and there is a potential problem due to holes and overlaps in the texture image when it is mapped into screen space. Forward mapping is like considering the texture map as a rubber sheet – stretching it in a way (determined by the parametrization) so that it sticks on the object surface thereby performing the normal object space to screen space transform.

8.1.1 Inverse mapping by bilinear interpolation

Although forward mapping is easy to understand, in practical algorithms inverse mapping is preferred and from now on we will only consider this strategy for polygon mesh objects. For inverse mapping it is convenient to consider a single (compound) transformation from two-dimensional screen space (x, y) to two-dimensional texture space (u, v). This is just an image warping operation and it can be modelled as a rational linear projective transform:

$$x = \frac{au + bv + c}{gu + hu + i} \qquad y = \frac{du + ev + f}{gu + hv + i} \tag{8.1}$$

This is, of course, a non-linear transformation as we would expect. Alternatively, we can write this in homogeneous coordinates as:

$$\begin{bmatrix} x' \\ y' \\ w \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} u' \\ v' \\ q \end{bmatrix}$$

where:

$$(x, y) = (x'/w, y'/w) \quad \text{and} \quad (u, v) = (u'/q, v'/q)$$

This is known as a rational linear transformation. The inverse transform – the one of interest to us in practice – is given by:

$$\begin{bmatrix} u' \\ v' \\ q \end{bmatrix} = \begin{bmatrix} \mathrm{A} & \mathrm{B} & \mathrm{C} \\ \mathrm{D} & \mathrm{E} & \mathrm{F} \\ \mathrm{G} & \mathrm{H} & \mathrm{I} \end{bmatrix} \begin{bmatrix} x' \\ y' \\ w \end{bmatrix}$$

$$= \begin{bmatrix} ei - fh & ch - bi & bf - ce \\ fg - di & ai - cg & cd - af \\ dh - eg & bg - ah & ae - bd \end{bmatrix} \begin{bmatrix} x' \\ y' \\ w \end{bmatrix}$$

Now recall that in most practical texture mapping applications we set up, during the modelling phase, an association between polygon mesh vertices and texture map coordinates. So, for example if we have the association for the four vertices of a quadrilateral we can find the nine coefficients $(a, b, c, d, e, f, g, h, i)$. We thus have the required inverse transform for any point within the polygon. This is done as follows. Return to the first half of Equation 8.1, the equation for x. Note that we can multiply top and bottom by an arbitrary non-zero scalar constant without changing the value of y, in effect we only have five degrees of freedom – not six – and because of this we can, without loss of generality set $i = 1$. Thus, in the overall transformation we only have 8 coefficients to determine and our quadrilateral-to-quadrilateral association will give a set of 8 equations in 8 unknowns which can be solved by any standard algorithm for linear equations – Gaussian elimination, for example. Full details of this procedure are given in Heckbert (1986).

A better practical alternative is to achieve the same effect by bilinear interpolation in screen space. So we interpolate the texture coordinates at the same time as interpolating lighting and depth. However, we note from the above that it is the homogeneous coordinates (u', v', q) that we have to interpolate, because the u and v do not change linearly with x and y.

Assuming vertex coordinate/texture coordinate for all polygons we consider each vertex to have homogeneous texture coordinates:

$$(u', v', q)$$

where:

$$u = u'/q$$
$$v = v'/q$$
$$q = 1/z$$

We interpolate using the normal bilinear interpolation scheme within the polygon (see Section 1.5) using these homogeneous coordinates as vertices to give (u', v', q') for each pixel then the required texture coordinates are given by:

$$(u, v) = u'/q, v'/q$$

Note that this costs two divides per pixel. For the standard incremental implementation of this interpolation process we need three gradients down each edge (in the current edge-pair) and three gradients for the current scan line.

8.1.2 Inverse mapping by using an intermediate surface

The previous method for mapping two-dimensional textures is now undoubtedly the most popular approach. The method we now describe can be used in applications where there is no texture coordinate–vertex coordinate correspondence. Alternatively it can be used as a pre-process to determine this correspondence and the first method then used during rendering.

Two-part texture mapping is a technique that overcomes the surface parametrization problem in polygon mesh objects by using an 'easy' intermediate surface onto which the texture is initially projected. Introduced by Bier and Sloan (1986), the method can also be used to implement environment mapping and is thus a method that unifies texture mapping and environment mapping.

The process is known as two-part mapping because the texture is mapped onto an intermediate surface before being mapped onto the object. The intermediate surface is, in general, non-planar but it possesses an analytic mapping function and the two-dimensional texture is mapped onto this surface without difficulty. Finding the correspondence between the object point and the texture point then becomes a three-dimensional to three-dimensional mapping.

The basis of the method is most easily described as a two-stage forward mapping process (Figure 8.4):

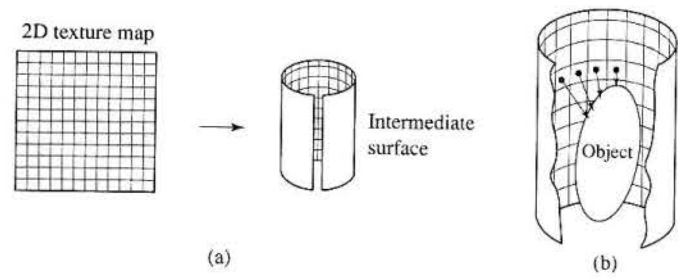


Figure 8.4
Two-stage mapping as a foward process. (a) *S* mapping. (b) *O* mapping.

(1) The first stage is a mapping from two-dimensional texture space to a simple three-dimensional intermediate surface such as a cylinder.

$$T(u, v) \rightarrow T'(x_i, y_i, z_i)$$

This is known as the *S* mapping.

(2) A second stage maps the three-dimensional texture pattern onto the object surface.

$$T'(x_i, y_i, z_i) \rightarrow O(x_w, y_w, z_w)$$

This is referred to as the *O* mapping.

These combined operations can distort the texture pattern onto the object in a 'natural' way, for example, one variation of the method is a 'shrinkwrap' mapping, where the planar texture pattern shrinks onto the object in the manner suggested by the eponym.

For the *S* mapping, Bier describes four intermediate surfaces: a plane at any orientation, the curved surface of a cylinder, the faces of a cube and the surface of a sphere. Although it makes no difference mathematically, it is useful to consider that $T(u, v)$ is mapped onto the interior surfaces of these objects. For example, consider the cylinder. Given a parametric definition of the curved surface of a cylinder as a set of points (θ, h), we transform the point (u, v) onto the cylinder as follows. We have:

$$S_{\text{cylinder}}: (\theta, h) \rightarrow (u, v)$$
$$= \left(\frac{r}{c}(\theta - \theta_0), \frac{1}{d}(h - h_0)\right)$$

where c and d are scaling factors and θ_0 and h_0 position the texture on the cylinder of radius r.

Various possibilities occur for the *O* mapping where the texture values for $O(x_w, y_w, z_w)$ are obtained from $T'(x_i, y_i, z_i)$, and these are best considered from a ray tracing point of view. The four *O* mappings are shown in Figure 8.5 and are:

(1) The intersection of the reflected view ray with the intermediate surface, T'. (This is, in fact, identical to environment mapping described in Section 8.6. The only difference between the general process of using this *O* mapping and environment mapping is that the texture pattern that is mapped onto the intermediate surface is a surrounding environment like a room interior.)

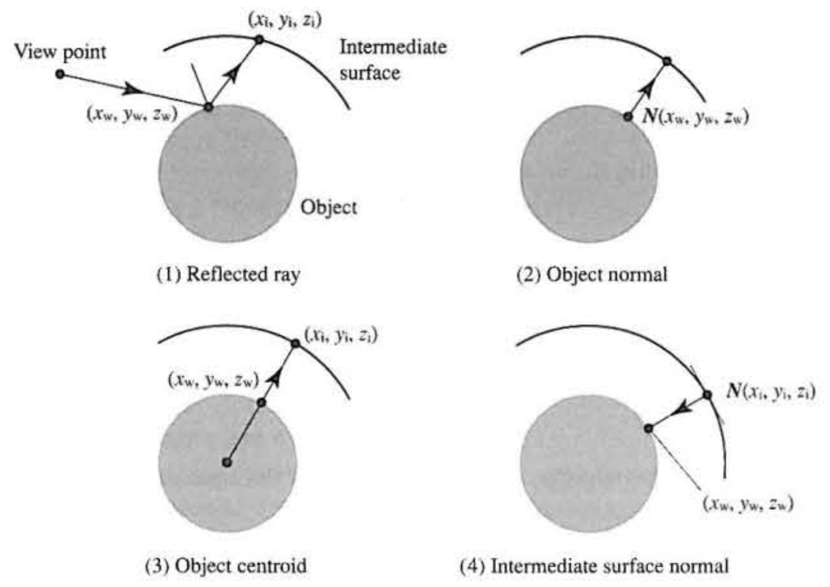


Figure 8.5 The four possible O mappings that map the intermediate surface texture T' onto the object

(2) The intersection of the surface normal at (x_w, y_w, z_w) with T'.

(3) The intersection of a line through (x_w, y_w, z_w) and the object centroid with T'.

(4) The intersection of the line from (x_w, y_w, z_w) to T' whose orientation is given by the surface normal at (x_i, y_i, z_i). If the intermediate surface is simply a plane then this is equivalent to considering the texture map to be a slide in a slide projector. A bundle of parallel rays of light from the slide projector impinges on the object surface. Alternatively it is also equivalent to three-dimensional texture mapping (see Section 8.7) where the field is defined by 'extruding' the two-dimensional texture map along an axis normal to the plane of the pattern.

Let us now consider this procedure as an inverse mapping process for the shrinkwrap case. We break the process into three stages (Figure 8.6).

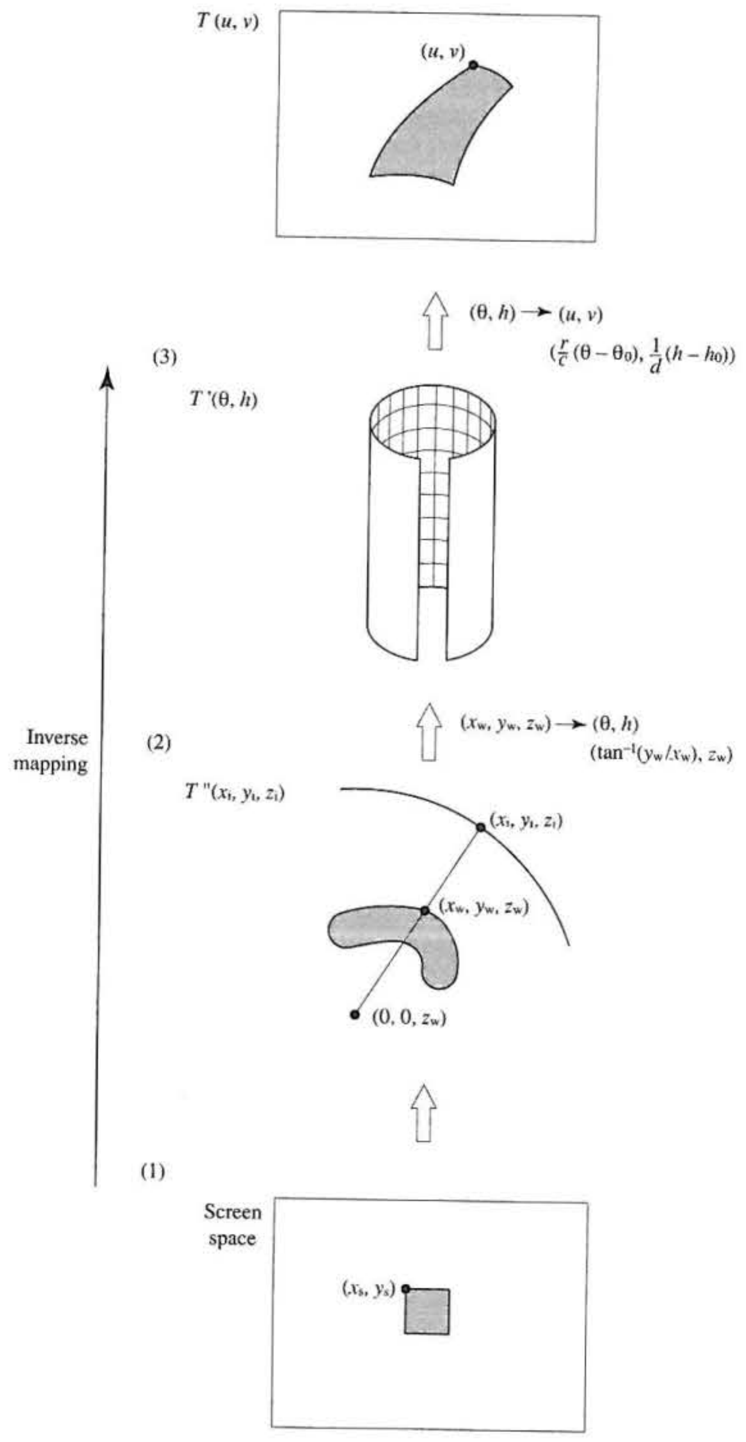
(1) Inverse map four pixel points to four points (x_w, y_w, z_w) on the surface of the object.

(2) Apply the O mapping to find the point (θ, h) on the surface of the cylinder. In the shrinkwrap case we simply join the object point to the centre of the cylinder and the intersection of this line with the surface of the cylinder gives us (x_i, y_i, z_i).

$x_w, y_w, z_w \rightarrow (\theta, h)$

$= (\tan^{-1}(y_w/x_w), z_w)$

(3) Apply the S mapping to find the point (u, v) corresponding to (θ, h).



$T(u, v)$
(u, v)
(3)
$(\theta, h) \rightarrow (u, v)$
$(\frac{r}{c}(\theta - \theta_0), \frac{1}{d}(h - h_0))$
$T'(\theta, h)$
Inverse mapping
(2)
$(x_w, y_w, z_w) \rightarrow (\theta, h)$
$(\tan^{-1}(y_w/x_w), z_w)$
$T''(x_i, y_i, z_i)$
(x_i, y_i, z_i)
(x_w, y_w, z_w)
$(0, 0, z_w)$
(1)
Screen space
(x_s, y_s)

Figure 8.6 Inverse mapping using the shrinkwrap method.

Figure 8.7 (Colour Plate) shows examples of mapping the same texture onto an object using different intermediate surfaces. The intermediate objects are a plane (equivalently no intermediate surface – the texture map is a plane), a cylinder and a sphere. The simple shape of the vase was chosen to illustrate the different distortions that each intermediate object produces. There are two points that can be made from these illustrations. First, you can choose an intermediate mapping that is appropriate to the shape of the object. A solid of revolution may be best suited, for example, to a cylinder. Second, although the method does not place any constraints on the shape of the object, the final visual effect may be deemed unsatisfactory. Usually what we mean by texture does not involve the texture pattern being subject to large geometric distortions. It is for this reason that many practical methods are interactive and involve some strategy like pre-distorting the texture map in two-dimensional space until its produces a good result when it is stuck onto the object.

8.2 Two-dimensional texture domain to bi-cubic parametric patch objects

If an object is a quadric or a cubic then surface parametrization is straightforward. In the previous section we used quadrics as intermediate surfaces exactly for this reason. If the object is a bi-cubic parametric patch, texture mapping is trivial since a parametric patch by definition already possesses (u, v) values everywhere on its surface.

The first use of texture in computer graphics was a method developed by Catmull. This technique applied to bi-cubic parametric patch models; the algorithm subdivides a surface patch in object space, and at the same time executes a corresponding subdivision in texture space. The idea is that the patch subdivision proceeds until it covers a single pixel (a standard patch rendering approach described in detail in Chapter 4). When the patch subdivision process terminates the required texture value(s) for the pixel is obtained from the area enclosed by the current level of subdivision in the texture domain. This is a straightforward technique that is easily implemented as an extension to a bi-cubic patch renderer. A variation of this method was used by Cook where object surfaces are subdivided into 'micro-polygons' and flat shaded with values from a corresponding subdivision in texture space.

An example of this technique is shown in Figure 8.8 (Colour Plate). Here each patch on the teapot causes subdivision of a single texture map, which is itself a rendered version of the teapot. For each patch, the u, v values from the parameter space subdivision are used to index the texture map whose u, v values also vary between 0 and 1. This scheme is easily altered to, say, map four patches into the entire texture domain by using a scale factor of two in the u, v mapping.

8.3 Billboards

Billboard is the name given to a technique where a texture map is considered as a three-dimensional entity and placed in the scene, rather than as a map that controls the colour over the surface of an object. It is a simple technique that utilizes a two-dimensional image in a three-dimensional scene by rotating the plane of the image so that it is normal to its viewing direction (the line from the view point to its position). The idea is illustrated in Figure 8.9. Probably the most common example of this is the image of a tree which is approximately cylindrically symmetric. Such objects are impossible to render in real time and the visual effect of this trick is quite convincing providing the view vector is close to the horizontal plane in scene space. The original two-dimensional nature of the object is hardly noticeable in the two-dimensional projection, presumably because we do not have an accurate internal notion of what the projection of the tree should look like anyway. The billboard is in effect a two-dimensional object which is rotated about its y axis (for examples like the tree) through an angle which makes it normal to the view direction and translated to the appropriate position in the scene. The background texels in the billboard are set to transparent.

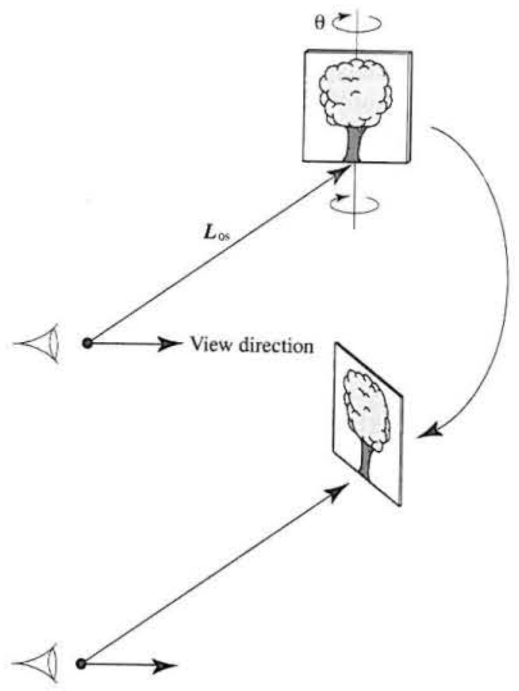


Figure 8.9
Providing the viewing direction is approximately parallel to the ground plane, objects like trees can be represented as a billboard and rotated about their y axis so that they are oriented normal to the L_{os} vector.

The modelling rotation for the billboard is given as:

$$\theta = \pi - \cos^{-1}(\boldsymbol{L}_{os} \cdot \boldsymbol{B}_n)$$

where:

$\boldsymbol{B}_n$ is the normal vector of the billboard, say (0,0,1)

$\boldsymbol{L}_{os}$ is the viewing direction vector from the view point to the required position of the billboard in world coordinates

Given θ and the required translation we can then construct a modelling transformation for the geometry of the billboard and transform it. Of course, this simple example will only work if the viewing direction is parallel or approximately parallel to the ground plane. When this is not true the two-dimensional nature of the billboard will be apparent.

Billboards are a special case of impostors or sprites which are essentially precomputed texture maps used to by-pass normal rendering when the view point is only changing slightly. These are described in detail in Chapter 14.

8.4 Bump mapping

Bump mapping, a technique developed by Blinn (1978), is an elegant device that enables a surface to appear as if it were wrinkled or dimpled without the need to model these depressions geometrically. Instead, the surface normal is angularly perturbed according to information given in a two-dimensional bump map and this 'tricks' a local reflection model, wherein intensity is a function mainly of the surface normal, into producing (apparent) local geometric variations on a smooth surface. The only problem with bump mapping is that because the pits or depressions do not exist in the model, a silhouette edge that appears to pass through a depression will not produce the expected cross-section. In other words the silhouette edge will follow the original geometry of the model.

It is an important technique because it appears to texture a surface in the normal sense of the word rather than modulating the colour of a flat surface. Figure 8.10 (Colour Plate) shows examples of this technique.

Texturing the surface in the rendering phase, without perturbing the geometry, by-passes serious modelling problems that would otherwise occur. If the object is polygonal the mesh would have to be fine enough to receive the perturbations from the texture map – a serious imposition on the original modelling phase, particularly if the texture is to be an option. Thus the technique converts a two-dimensional height field $B(u, v)$, called the bump map, and which represents some desired surface displacement, into appropriate perturbations of the local surface normal. When this surface normal is used in the shading equation the reflected light calculations vary as if the surface had been displaced.

Consider a point $\boldsymbol{P}(u, v)$ on a (parameterized) surface corresponding to $B(u, v)$. We define the surface normal at the point to be:

$$\boldsymbol{N} = \frac{\partial \boldsymbol{P}}{\partial u} \times \frac{\partial \boldsymbol{P}}{\partial v}$$

$$= \boldsymbol{P}_u \times \boldsymbol{P}_v$$

where $\boldsymbol{P}_u$ and $\boldsymbol{P}_v$ are the partial derivatives lying in the tangent plane to the surface at point $\boldsymbol{P}$. What we want to do is to have the same effect as displacing the point $\boldsymbol{P}$ in the direction of the surface normal at that point by an amount $B(u, v)$ – a one-dimensional analogue is shown in Figure 8.11. That is:

$$\boldsymbol{P}'(u, v) = \boldsymbol{P}(u, v) + B(u, v)\boldsymbol{N}$$

Locally the surface would not now be as smooth as it was before because of this displacement and the normal vector $\boldsymbol{N}'$ to the 'new' surface is given by differentiating this equation:

$$\boldsymbol{N}' = \boldsymbol{P}'_u + \boldsymbol{P}'_v$$

$$\boldsymbol{P}'_u = \boldsymbol{P}_u + B_u\boldsymbol{N} + B(u, v)\boldsymbol{N}_u$$

$$\boldsymbol{P}'_v = \boldsymbol{P}_v + B_v\boldsymbol{N} + B(u, v)\boldsymbol{N}_v$$

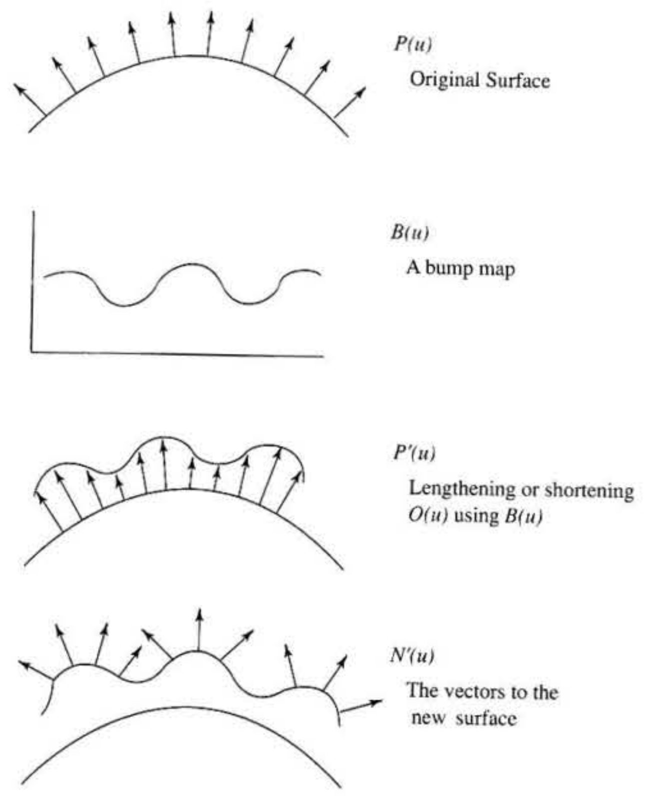


Figure 8.11
A one-dimensional example of the stages involved in bump mapping (after Blinn (1978)).

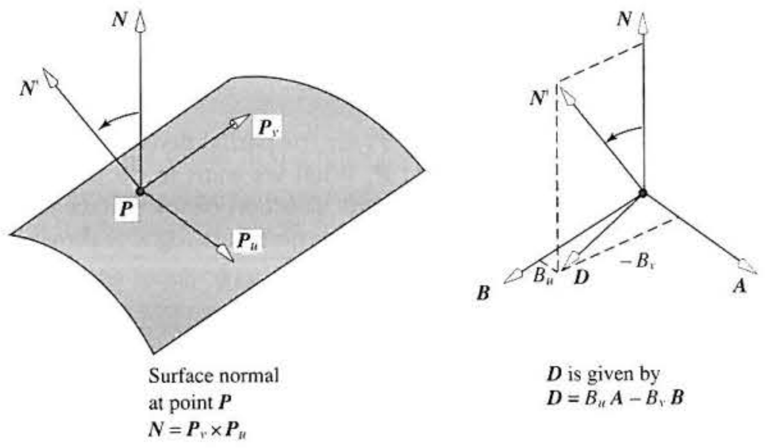


Figure 8.12 Geometric interpretation of bump mapping.

If B is small we can ignore the final term in each equation and we have:

$$\boldsymbol{N}' = \boldsymbol{N} + B_u \boldsymbol{N} \times \boldsymbol{P}_v + B_v \boldsymbol{P}_u \times \boldsymbol{N}$$

or

$$\begin{aligned}\boldsymbol{N}' &= \boldsymbol{N} + B_u \boldsymbol{N} \times \boldsymbol{P}_v - B_v \boldsymbol{N} \times \boldsymbol{P}_u \\ &= \boldsymbol{N} + (B_u \boldsymbol{A} - B_v \boldsymbol{B}) \\ &= \boldsymbol{N} + \boldsymbol{D}\end{aligned}$$

Then $\boldsymbol{D}$ is a vector lying in the tangent plane that 'pulls' $\boldsymbol{N}$ into the desired orientation and is calculated from the partial derivatives of the bump map and the two vectors in the tangent plane (Figure 8.12).

8.4.1 A multi-pass technique for bump mapping

For polygon mesh objects McReynolds and Blythe (1997) define a multi-pass technique that can exploit standard texture mapping hardware facilities. To do this they split the calculation into two components as follows. The final intensity value is proportional to $\boldsymbol{N}' \cdot \boldsymbol{L}$ where:

$$\boldsymbol{N}' \cdot \boldsymbol{L} = \boldsymbol{N} \cdot \boldsymbol{L} + \boldsymbol{D} \cdot \boldsymbol{L}$$

The first component is the normal Gouraud component and the second component is found from the differential coefficient of two image projections formed by rendering the surface with the height field as a normal texture map. To do this it is necessary to transform the light vector into tangent space at each vertex of the polygon. This space is defined by $\boldsymbol{N}$, $\boldsymbol{B}$ and $\boldsymbol{T}$, where:

$\boldsymbol{N}$ is the vertex normal
$\boldsymbol{T}$ is the direction of increasing u (or v) in the object space coordinate system
$\boldsymbol{B} = \boldsymbol{N} \times \boldsymbol{T}$

The normalized components of these vectors defines the matrix that transforms points into tangent space:

$$\boldsymbol{L}_{TS} = \begin{bmatrix} T_X & T_Y & T_Z & 0 \\ B_X & B_Y & B_Z & 0 \\ N_X & N_Y & N_Z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \boldsymbol{L}$$

The algorithm is as follows:

(1) The object is rendered using a normal renderer with texture mapping facilities. The texture map used is the bump map or height field.

(2) $\boldsymbol{T}$ and $\boldsymbol{B}$ are found at each vertex and the light vector transformed into tangent space.

(3) A second image is created in the same way but now the texture/vertex correspondence is shifted by small amounts in the direction of the X, Y components of $\boldsymbol{L}_{TS}$. We now have two image projections where the height field or the bump map has been mapped onto the object and shifted with respect to the surface. If we subtract these two images we get the differential coefficient which is the required term $\boldsymbol{D} \cdot \boldsymbol{L}$. (Finding the differential coefficient of an image by subtraction is a standard image processing technique – see, for example, Watt and Policarpo (1998)).

(4) The object is rendered in the normal manner *without* any texture and this component is added to the subtrahend calculated in step (3) to give the final bump-mapped image.

Thus we replace the explicit bump mapping calculations with two texture mapped rendering passes, an image subtract, a Gouraud shading pass then an image added to get the final result.

8.4.2 A pre-calculation technique for bump mapping

Tangent space can also be used to facilitate a pre-calculation technique as proposed by Peercy *et al.* (1997). This depends on the fact that the perturbed normal $\boldsymbol{N}'_{TS}$ in tangent space is a function only of the surface itself and the bump map. Peercy *et al.* define this normal at each vertex in terms of three pre-calculated coefficients.

It can be shown (Peercy *et al.* 1997) that the perturbed normal vector on tangent space is given by:

$$\boldsymbol{N}'_{TS} = \frac{a, b, c}{(a^2 + b^2 + c^2)^{1/2}}$$

where:

$$a = -B_u(\boldsymbol{B}.\boldsymbol{P}_v)$$

$$b = -(B_v|\boldsymbol{P}_u| - B_u(\boldsymbol{T}.\boldsymbol{P}_v))$$

$$c = |\boldsymbol{P}_u \times \boldsymbol{P}_v|$$

For each point in the bump map these points can be pre-computed and a map of perturbed normals is stored for use during rendering instead of the bump map.

8.5 Light maps

Light maps are an obvious extension to texture maps that enable lighting to be pre-calculated and stored as a two-dimensional texture map. We sample the reflected light over a surface and store this in a two-dimensional map. Thus shading reduces to indexing into a light map or a light modulated texture map. An advantage of the technique is that there is no restriction on the complexity of the rendering method used in the pre-calculation – we could, for example, use radiosity or any view-independent global illumination method to generate the light maps.

In principle light maps are similar to environment maps (see Section 8.6). In environment mapping we cache, in a two-dimensional map, all the illumination incident at a single point in the scene. With light maps we cache the reflected light from every surface in the scene in a set of two-dimensional maps.

If an accurate pre-calculation method is used then we would expect the technique to produce better quality shading and be faster than Gouraud interpolation. This means that we can incorporate shadows in the final rendering. The obvious disadvantage of the technique is that for moving objects we can only invoke a very simple lighting environment (diffuse shading with the light source at infinity). A compromise is to use dynamic shading for moving objects and assume that they do not interact, as far as shading is concerned, with static objects shaded with a light map.

Light maps can either be stored separately from texture maps, or the object's texture map can be pre-modulated by the light map. If the light map is kept as a separate entity then it can be stored at a lower resolution than the texture map because view-independent lighting, except at shadow edges, changes more slowly than texture detail. It can also be high-pass filtered which will ameliorate effects such as banding in the final image and also has the benefit of blurring shadow edges (in the event that a hard-edged shadow generation procedure has been used).

If an object is to receive a texture then we can modulate the brightness of the texture during the modelling phase so that it has the same effect as if the (unmodulated) texture colours were injected into, say, a Phong shading

equation. This is called surface caching because it stores the final value required for the pixel onto which a surface point projects and because texture caching hardware is used to implement it. If this strategy is employed then the texture mapping transform and the transform that maps light samples on the surface of the object into a light map should be the same.

Light maps were first used in two-pass ray tracing (see Section 10.7) and are also used in Ward's (1994) RADIANCE renderer. Their motivation in these applications was to cache diffuse illumination and to enable the implementation of a global illumination model that would work in a reasonable time. Their more recent use in games engines has, of course, been to facilitate shading in real time.

The first problem with light maps is how do we sample and store, in a two-dimensional array, the calculated reflected light across the face of a polygon in three-dimensional space. In effect this is the reverse of texture mapping where we need a mapping from two-dimensional space into three-dimensional object space. Another problem concerns economy. For scenes of any complexity it would clearly be uneconomical to construct a light map for each polygon – rather we require many polygons to share a single light map.

Zhukov *et al.* (1998) approach the three-dimensional sampling problem by organizing polygons into structures called 'polypacks'. Polygons are projected into the world coordinate planes and collected into polypacks if their angle with a coordinate plane does not exceed some threshold (so that the maximal projection plane is selected for a polygon) and if their extent does not overlap in the projection. The world space coordinate planes are subdivided into square cells (the texels or 'lumels') and back projected onto the polygon. The image of a square cell on a polygon is a parallelogram (whose larger angle ≤ 102°). These are called patches and are the subdivided polygon elements for which the reflected light is calculated. This scheme thus samples every polygon with almost square elements storing the result in the light map (Figure 8.13).

These patches form a subdivision of the scene sufficient for the purpose of generating light maps and a single light intensity for each patch can be calculated using whatever algorithm the application demands (for example Phong shading or radiosity). After this phase is complete there exists a set of (parallelogram-shaped) samples for each polygon. These then have to be 'stuffed'

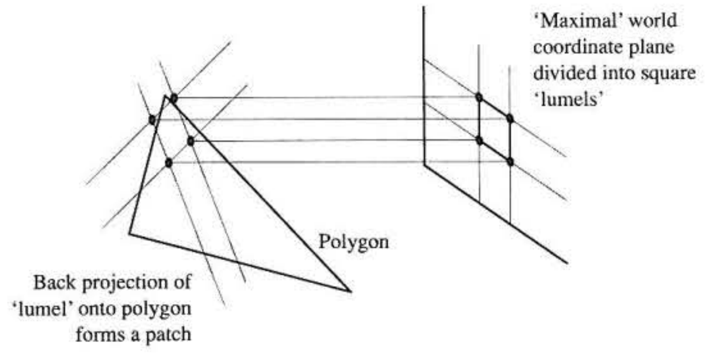

'Maximal' world coordinate plane divided into square 'lumels'
Polygon
Back projection of 'lumel' onto polygon forms a patch

Figure 8.13 Forming a light map in the 'maximal' world coordinate plane.

into the minimum number of two-dimensional light maps in each coordinate plane. Zhukov *et al.* (1998) do this by using a 'first hit decreasing' algorithm which first sorts each polygon group by the number of texels.

Another problem addressed by the authors is that the groups of samples corresponding to a polygon have to be surrounded by 'sand' texels. These are supplementary texels that do not belong to a face group but are used when bilinear interpolation is used in conjunction with a light map and prevent visual lighting artefacts appearing at the edges of polygons. Thus each texture map consists of a mixture of light texels, sand texels and unoccupied texels. Zhukov *et al.* (1998) report that for a scene consisting of 24 000 triangles (700 patches) 14 light maps of 256 × 256 texels were produced which exhibited a breakdown of texels as: 75% light texels, 15% sand texels and 10% unoccupied texels.

A direct scheme for computing light maps for scenes made up of triangles and for which we already have vertex/texture coordinate association is to use this correspondence to derive an affine transformation between texture space and object space and then use this transformation to sample the light across the face of a triangle. The algorithm is then driven from the texture map space (by scan-converting the polygon projection in texture space) and for each texel finding its corresponding point or projection on the object surface from:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

where (x, y, z) is the point on the object corresponding to the texel (u, v). This transformation can be seen as a linear transformation in three-dimensional space with the texture map embedded in the $z = 1$ plane. The coefficients are found from the vertex/texture coordinate correspondence by inverting the U matrix in:

$$\begin{bmatrix} x_0 & x_1 & x_2 \\ y_0 & y_1 & y_2 \\ z_0 & z_1 & z_2 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} u_0 & u_1 & u_2 \\ v_0 & v_1 & v_2 \\ 1 & 1 & 1 \end{bmatrix}$$

writing this as:

$$X = AU$$

we have:

$$A = XU^{-1}$$

The inverse U^{-1} is guaranteed to exist providing the three points are non-collinear. Note that in terms of our treatment in Section 8.1 this is a forward mapping from texture space to object space. Examples of a scene lit using this technique are shown in Figure 8.14 (Colour Plate).

8.6 Environment or reflection mapping

Originally called reflection mapping and first suggested by Blinn and Newell (1976), environment mapping was consolidated into mainline rendering techniques in an important paper by Greene (1986). Environment maps are a shortcut to rendering shiny objects that reflect the environment in which they are placed. They can approximate the quality of a ray tracer for specular reflections and do this by reducing the problem of following a reflected view vector to indexing into a two-dimensional map which is no different from a conventional texture map. Thus processing costs that would be incurred in ray tracing programs are regulated to the (off-line) construction of the map(s). In this sense it is a classic partial off-line or pre-calculation technique like pre-sorting for hidden surface removal. An example of a scene and its corresponding (cubic) environment map is shown in Figure 18.8.

The disadvantages of environment mapping are:

- It is (geometrically) correct only when the object becomes small with respect to the environment that contains it. This effect is usually not noticeable in the sense that we are not disturbed by 'wrong' reflections in the curved surface of a shiny object. The extent of the problem is shown in Figure 18.9 which shows the same object ray traced and environment mapped.
- An object can only reflect the environment – not itself – and so the technique is 'wrong' for concave objects. Again this can be seen in Figure 18.9 where the reflection of the spout is apparent in the ray traced image.
- A separate map is required for each object in the scene that is to be environment mapped.
- In one common form of environment mapping (sphere mapping) a new map is required whenever the view point changes.

In this section we will examine three methods of environment mapping which are classified according to the way in which the three-dimensional environment information is mapped into two-dimensions. These are cubic, latitude–longitude and sphere mapping. (Latitude–longitude is also a spherical mapping but the term sphere mapping is now applied to the more recent form.) The general principles are shown in Figure 8.15. Figure 8.15(a) shows the conventional ray tracing paradigm which we replace with the scheme shown in Figure 8.15(b). This involves mapping the reflected view vector into a two-dimensional environment map. We calculate the reflected view vector as (Section 1.3.5):

$$\boldsymbol{R}_v = 2(\boldsymbol{N}\cdot\boldsymbol{V})\boldsymbol{N} - \boldsymbol{V} \qquad [8.2]$$

Figure 8.15(c) shows that, in practice, for a single pixel we should consider the reflection beam, rather than a single vector, and the area subtended by the beam in the map is then filtered for the pixel value. A reflection beam originates either

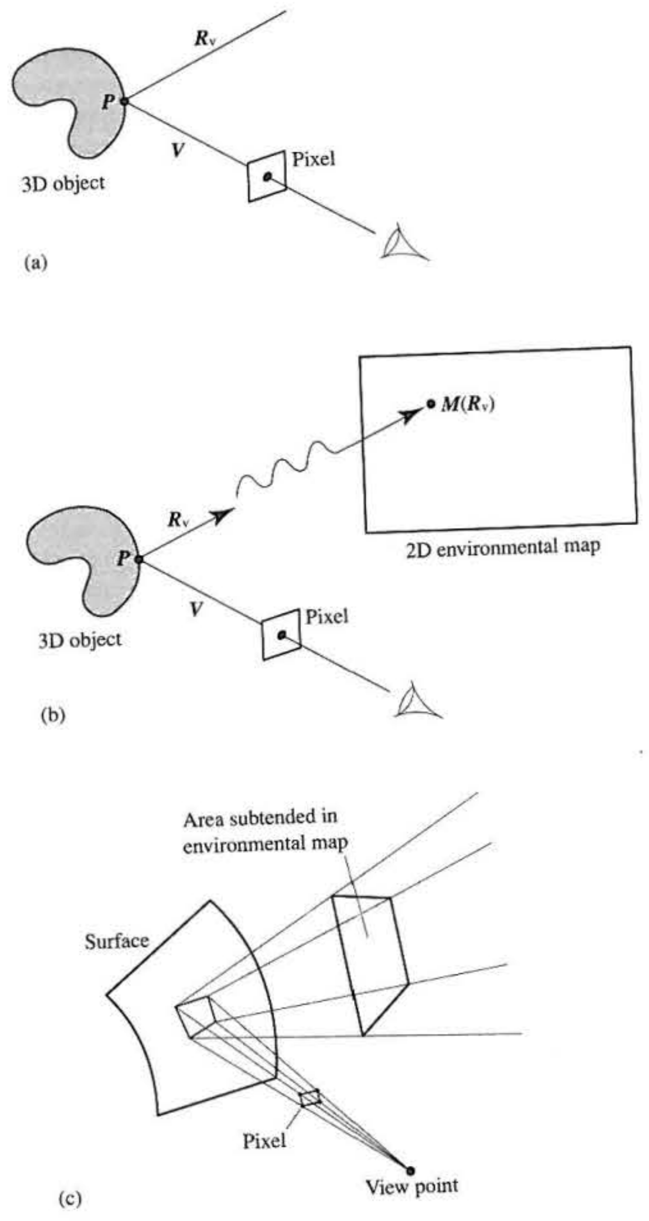


Figure 8.15
Environment mapping
(a) The ray tracing model – that part of the environment reflected at point ***P*** is determined by reflecting the view ray $\boldsymbol{R}_v$. (b) We try to achieve the same effect as in (a) by using a function of $\boldsymbol{R}_v$ to index into a two-dimensional map. (c) A pixel subtends a reflection beam.

from four pixel corners if we are indexing the map for each pixel, or from polygon vertices if we are using a fast (approximate) scheme. An important point to note here is that the area intersected in the environment map is a function of the curvature of the projected pixel area on the object surface. However, because we are now using texture mapping techniques we can employ pre-filtering anti-aliasing methods (see Section 8.8).

In real time polygon mesh rendering, we can calculate reflected view vectors only at the vertices and use linear interpolation as we do in conventional texture mapping. Because we expect to see fine detail in the resulting image, the quality of this approach depends strongly on the polygon size.

In effect an environment map caches the incident illumination from all directions at a single point in the environment with the object that is to receive the mapping removed from the scene. Reflected illumination at the surface of an object is calculated from this incident illumination by employing the aforementioned geometric approximation – that the size of the object itself can be considered to approach the point and a simple BRDF which is a perfect specular term – the reflected view vector. It is thus a view-independent pre-calculation technique.

8.6.1 Cubic mapping

As we have already implied, environment mapping is a two-stage process that involves – as a pre-process – the construction of the map. Cubic mapping is popular because the maps can easily be constructed using a conventional rendering system. The environment map is in practice six maps that form the surfaces of a cube (Figure 8.16). An example of an environment map is shown in Figure 18.8. The view point is fixed at the centre of the object to receive the environment map, and six views are rendered. Consider a view point fixed at the centre of a room. If we consider the room to be empty then these views would contain the

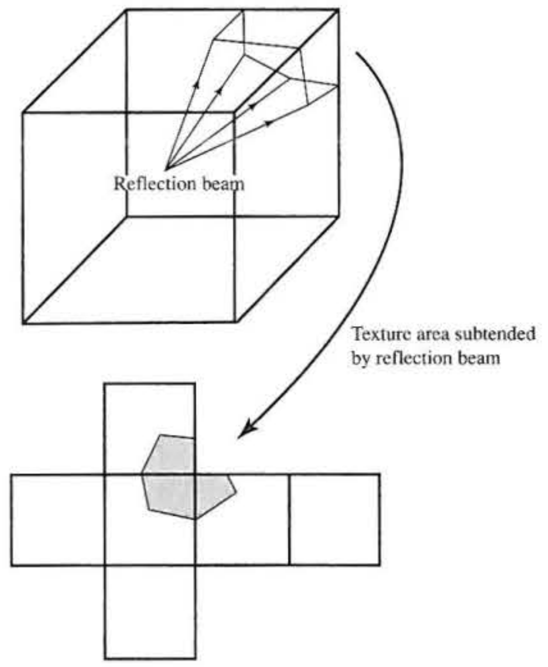


Figure 8.16
Cubic environment mapping: the reflection beam can range over more than one map.

four walls and the floor and ceiling. One of the problems of a cubic map is that if we are considering a reflection beam formed by pixel corners, or equivalently by the reflected view vectors at a polygon vertex, the beam can index into more than one map (Figure 18.16). In that case the polygon can be subdivided so that each piece is constrained to a single map.

With cubic maps we need an algorithm to determine the mapping from the three-dimensional view vector into one or more two-dimensional maps. (With the techniques described in the next section this mapping algorithm is replaced by a simple calculation.) If we consider that the reflected view vector is in the same coordinate frame as the environment map cube (the case if the view were constructed by pointing the (virtual or real) camera along the world axes in both directions), then the mapping is as follows.

For a single reflection vector:

(1) Find the face it intersects – the map number. This involves a simple comparison of the components of the normalized reflected view vector against the (unit) cube extent which is centred on the origin.

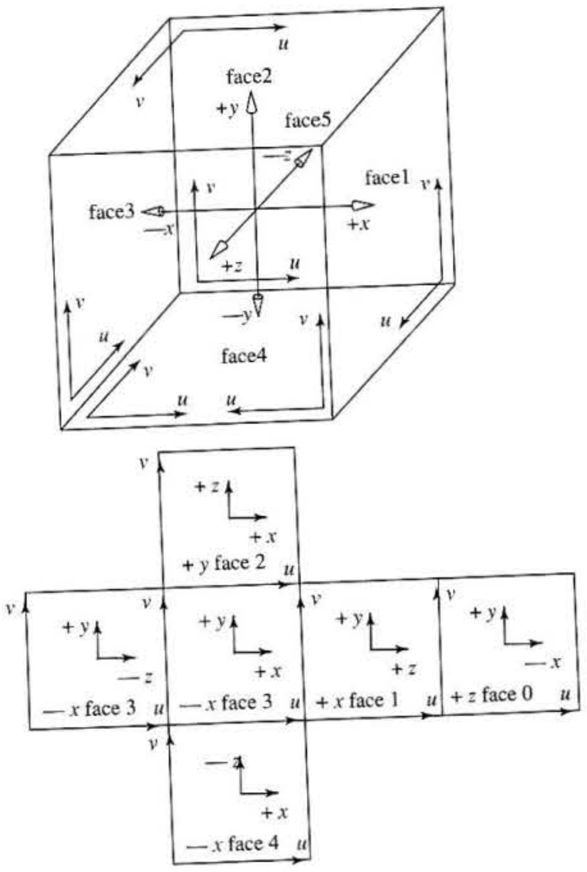


Figure 8.17
Cubic environment map convention.

(2) Map the components into (u, v) coordinates. For example, a point (x, y, z) intersecting the face normal to the negative z axis is given by:

$u = x + 0.5$
$v = -z + 0.5$

for the convention scheme shown in Figure 8.17.

One of the applications of cubic environment maps (or indeed any environment map method) that became popular in the 1980s is to 'matte' an animated computer graphics object into a real environment. In that case the environment map is constructed from photographs of a real environment and the (specular) computer graphics object can be matted into the scene, and appear to be part of it as it moves and reflects its surroundings.

8.6.2 Sphere mapping

The first use of environment mapping was by Blinn and Newell (1976) wherein a sphere rather than a cube was the basis of the method used. The environment map consisted of a latitude–longitude projection and the reflected view vector, R_v, was mapped into (u, v) coordinates as:

$$u = \frac{1}{2}\left(1+\frac{1}{\pi}\tan^{-1}\left(\frac{R_{vy}}{R_{vx}}\right)\right) \quad -\pi < \tan^{-1} < \pi$$

$$v = \frac{R_{vz}+1}{2}$$

The main problem with this simple technique is the singularities at the poles. In the polar area small changes in the direction of the reflection vector produce large changes in (u, v) coordinates. As $R_{vz} \to \pm 1$, both R_{vx} and $R_{vy} \to 0$ and R_{vy}/R_{vx} becomes ill-defined. Equivalently, as $v \to 1$ or 0 the behaviour of u starts to break down causing visual disturbances on the surface. This can be ameliorated by modulating the horizontal resolution of the map with sin θ (where θ is the elevation angle in polar coordinates).

An alternative sphere mapping form (Haeberli and Segal 1993; Miller *et al.* 1998) consists of a circular map which is the orthographic projection of the reflection of the environment as seen in the surface of a perfect mirror sphere (Figure 8.18). Clearly such a map can be generated by ray tracing from the view plane. (Alternatively a photograph can be taken of a shiny sphere.) Although the map caches the incident illumination at the reference point by using an orthographic projection it can be used to generate, to within the accuracy of the process, a normal perspective projection.

To generate the map we proceed as follows. We trace a parallel ray bundle – one ray for each texel (u, v) and reflect each ray from the sphere. The point on the sphere at the point hit by the ray from (u, v) is $\boldsymbol{P}$, where:

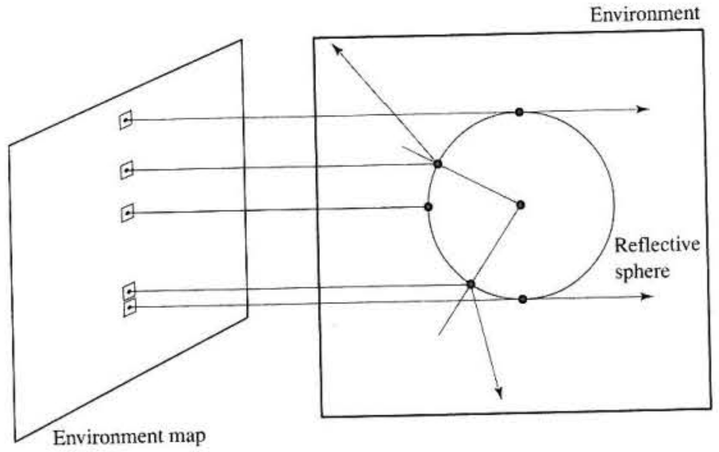


Figure 8.18
Constructing a spherical map by ray tracing from the map texels onto a reflective sphere.

$\boldsymbol{P}_x = u \qquad \boldsymbol{P}_y = v$
$\boldsymbol{P}_z = (1.0 - \boldsymbol{P}_x^2 - \boldsymbol{P}_y^2)^{1/2}$

This is also the normal to the sphere at the hit point and we can compute the reflected vector using Equation 8.2.

To index into the map we reflect the view vector from the object (either for each pixel or for each polygon vertex) and calculate the map coordinates as:

$$u = \frac{R_x}{m} + \frac{1}{2}$$

$$v = \frac{R_y}{m} + \frac{1}{2}$$

where:

$$m = 2(R_x^2 + R_y^2 + (R_z + 1)^2)^{1/2}$$

8.6.3 Environment mapping: comparative points

Sphere mapping overcomes the main limitation of cubic maps which require, in general, access to a number of the face maps, and is to be preferred when speed is important. However, both types of sphere mapping suffer more from non-uniform sampling than cubic mapping. Refer to Figure 8.19 which attempts to demonstrate this point. In all three cases we consider that the environment map is sampling incoming illumination incident on the surface of the unit sphere. The illustration shows the difference between the areas on the surface of the sphere sampled by a texel in the environment map. Sampling only approaches uniformity when the viewing direction during the rendering phase aligns with the viewing direction from which the map was computed. For this reason this type of spherical mapping is considered to be view dependent and a new map has to be computed when the view direction changes.

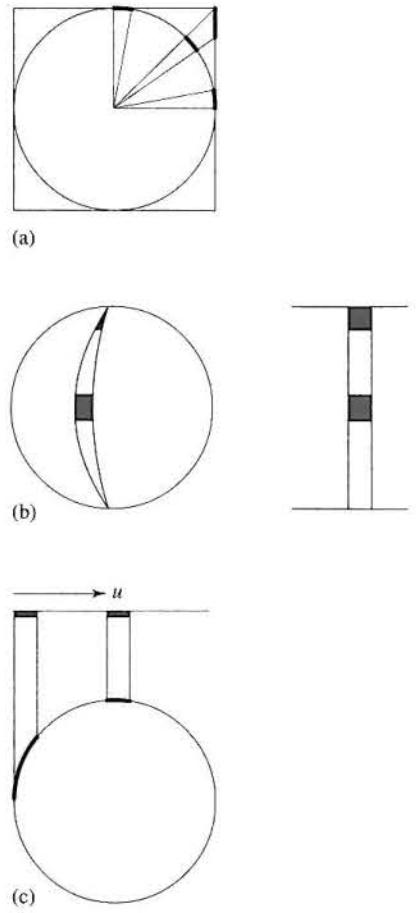


Figure 8.19
Sampling the surface of a sphere. (a) Cubic perspective: under-sampling at the centre of the map (equator and meridian) compared to the corners. (b) Mercator or latitude–longitude: severe over-sampling at edges of the map in the *v* direction (poles). (c) Orthographic: severe under-sampling at the edges of the map in the *u* direction (equator).

8.6.4 Surface properties and environment mapping

So far we have restricted the discussion to geometry and assumed that the object which is environment mapped possesses a perfect mirror surface and the map is indexed by a single reflected view ray. What if we want to use objects with reflecting properties other than that of a perfect mirror. Using the normal Phong local reflection model, we can consider two components – a diffuse component plus a specular component – and construct two maps. The diffuse map is indexed by the surface normal at the point of interest and the specular map is indexed by the reflected view vector. The relative contribution from each map is determined by diffuse and specular reflection coefficients just as in standard Phong shading. This enables us to render objects as if they were Phong shaded but with the addition of reflected environment detail which can be blurred to simulate a non-smooth surface. (Note that this approximates an effect that would otherwise have to be rendered using distributed ray tracing.)

This technique was first reported by Miller and Hoffman (1984). (This reference is to SIGGRAPH course notes. These, particularly the older ones, are generally unavailable and we only refer to them if the material does not, as far as we know, appear in any other publication) and it is their convention that we follow here. Both the diffuse and specular maps are generated by processing the environment map. Thus we can view the procedure as a two-step process where the first step – the environment map – encodes the illumination at a point due to the scene with the object removed and the second step filters the map to encode information about the surface of the object.

Miller and Hoffman (1984) generate the diffuse map from the following definition:

$$D(\boldsymbol{N}) = \frac{\sum_{L} I(\boldsymbol{L}) \times \text{Area}(\boldsymbol{L}) \times f_d\,(\boldsymbol{N}\cdot\boldsymbol{L})}{4\pi}$$

where:

$\boldsymbol{N}$ is the surface normal at the point of interest

$I(\boldsymbol{L})$ is the environment map as a function of $\boldsymbol{L}$ the incident direction to which the entry I in the map corresponds

Area is the area on the surface of the unit sphere associated with $\boldsymbol{L}$

f_d is the diffuse convolution function:

$f_d(x) = k_d\, x$ for $x > 0$ and $f_d(x) = 0$ for $x \le 0$

k_d is the diffuse reflection coefficient that weights the contribution of $D(\boldsymbol{N})$ in summing the diffuse and specular contributions

Thus for each value of $\boldsymbol{N}$ we sum over all values of $\boldsymbol{L}$ the area-weighted dot product or Lambertian term.

The specular map is defined as:

$$S(\boldsymbol{R}) = \frac{\sum_{L} I(\boldsymbol{L}) \times \text{Area}(\boldsymbol{L}) \times f_s\,(\boldsymbol{R}\cdot\boldsymbol{L})}{4\pi}$$

where:

$\boldsymbol{R}$ is the reflected view vector

f_s is the specular convolution function;

$f_s(x) = k_s\, x^n$ for $x > 0$ and $f_s(x) = 0$ for $x \le 0$

k_s is the specular reflection coefficient

(Note that if f_s is set to unity the surface is a perfect mirror and the environment map is unaltered.)

The reflected intensity at a surface point is thus:

$D(\boldsymbol{N}) + S(\boldsymbol{R})$

8.7 Three-dimensional texture domain techniques

We have seen in preceding sections that there are many difficulties associated with mapping a two-dimensional texture onto the surface of a three-dimensional object. The reasons for this are:

(1) Two-dimensional texture mapping based on a surface coordinate system can produce large variations in the compression of the texture that reflect a corresponding variation in the curvature of the surface.

(2) Attempting to continuously texture map the surface of an object possessing a non-trivial topology can quickly become very awkward. Textural continuity across surface elements that can be of a different type and can connect together in any ad hoc manner is problematic to maintain.

Three-dimensional texture mapping neatly circumvents these problems since the only information required to assign a point a texture value is its position in space. Assigning an object a texture just involves evaluating a three-dimensional texture function at the surface points of the object. A fairly obvious requirement of this technique is that the three-dimensional texture field is procedurally generated. Otherwise the memory requirements, particularly if three-dimensional mip-mapping is used, become exorbitant. Also, it is inherently inefficient to construct an entire cubic field of texture when we only require these values at the surface of the object.

Given a point (x, y, z) on the surface of an object, the colour is defined as $T(x, y, z)$, where T is the value of texture field. That is, we simply use the identity mapping (possibly in conjunction with a scaling):

$u = x \quad v = y \quad w = z$

where:

(u, v, w) is a coordinate in the texture field

This can be considered analogous to actually sculpting or carving an object out of a block of material. The colour of the object is determined by the intersection of its surface with the texture field. The method was reported simultaneously by Perlin (1985) and Peachey (1985) wherein the term 'solid texture' was coined.

The disadvantage of the technique is that although it eliminates mapping problems, the texture patterns themselves are limited to whatever definition that you can think up. This contrasts with a two-dimensional texture map; here any texture can be set up by using, say, a frame-grabbed image from a television camera.

8.7.1 Three-dimensional noise

A popular class of procedural texturing techniques all have in common the fact that they use a three-dimensional, or spatial, noise function as a basic modelling primitive. These techniques, the most notable of which is the simulation of turbulence, can produce a surprising variety of realistic, natural-looking texture effects. In this section we will concern ourselves with the issues involved in the algorithmic generation of the basic primitive – solid noise.

Perlin (1985) was the first to suggest this application of noise, defining a function *noise*() that takes a three-dimensional position as its input and returns a single scalar value. This is called model-directed synthesis – we evaluate the noise function only at the point of interest. Ideally the function should possess the following three properties:

(1) Statistical invariance under rotation.

(2) Statistical invariance under translation.

(3) A narrow bandpass limit in frequency.

The first two conditions ensure that the noise function is controllable – that is, no matter how we move or orientate the noise function in space, its general appearance is guaranteed to stay the same. The third condition enables us to sample the noise function without aliasing. Whilst an insufficiently sampled noise function may not produce noticeable defects in static images, if used in animation applications, incorrectly sampled noise will produce a shimmering or bubbling effect.

Perlin's method of generating noise is to define an integer lattice, or a set of points in space, situated at locations (i, j, k) where i, j and k are all integers. Each point of the lattice has a random number associated with it. This can be done either by using a simple look-up table or, as Perlin (1985) suggests, via a hashing function to save space. The value of the noise function, at a point in space coincident with a lattice point, is just this random number. For points in space not on the lattice – in general (u, v, w) – the noise value can be obtained by linear interpolation from the nearby lattice points. If, using this method, we generate a solid noise function $T(u, v, w)$ then it will tend to exhibit directional (axis aligned) coherences. These can be ameliorated by using cubic interpolation but this is far more expensive and the coherences still tend to be visible. Alternative noise generation methods that eliminate this problem are to be found in Lewis (1989); however, it is worth bearing in mind that the entire solid noise function is sampled by the surface and usually undergoes a transformation (it is modulated, for example, to simulate turbulence) and this in itself may be enough to eliminate the coherences.

8.7.2 Simulating turbulence

A single piece of noise can be put to use to simulate a remarkable number of effects. By far the most versatile of its applications is the use of the so-called turbulence function, as defined by Perlin, which takes a position x and returns a turbulent scalar value. It is written in terms of the progression, a one-dimensional version of which would be defined as:

$$\text{turbulence}(x) = \sum_{i=0}^{k} \text{abs}\left(\frac{\text{noise}\,(2^i x)}{2^i}\right)$$

The summation is truncated at k which is the smallest integer satisfying:

$$\frac{1}{2^{k+1}} < \text{the size of a pixel}$$

The truncation band limits the function ensuring proper anti-aliasing. Consider the difference between the first two terms in the progression, noise (x) and noise $(2x)/2$. The noise function in the latter term will vary twice as fast as the first – it has twice the frequency – and will contain features that are half the size of the first. Moreover, its contribution to the final value for the turbulence is also scaled by one-half. At each scale of detail the amount of noise added into the series is proportional to the scale of detail of the noise and inversely proportional to the frequency of the noise. This is self-similarity and is analogous to the self-similarity obtained through fractal subdivision, except that this time the subdivision drives not displacement, but octaves of noise, producing a function that exhibits the same noisy behaviour over a range of scales. That this function should prove so useful is best seen from the point of view of signal analysis, which tells us that the power spectrum of *turbulence*() obeys a $1/f$ power law, thereby loosely approximating the $1/f^2$ power law of Brownian motion.

The turbulence function in isolation only represents half the story, however. Rendering the turbulence function directly results in a homogeneous pattern that could not be described as naturalistic. This is due to the fact that most textures which occur naturally, contain some non-homogeneous structural features and so cannot be simulated by turbulence alone. Take marble, for example, which has easily distinguished veins of colour running through it that were made turbulent before the marble solidified during an earlier geological era. In the light of this fact we can identify two distinct stages in the process of simulating turbulence, namely:

(1) Representation of the basic, first order, structural features of a texture through some basic functional form. Typically the function is continuous and contains significant variations in its first derivatives.

(2) Addition of second and higher order detail by using turbulence to perturb the parameters of the function.

The classic example, as first described by Perlin, is the turbulation of a sine wave to give the appearance of marble. Unperturbed, the colour veins running through the marble are given by a sine wave passing through a colour map. For a sine wave running along the x axis we write:

$$\text{marble}(x) = \text{marble\_colour}\,(\sin(x))$$

The colour map *marble_colour*() maps a scalar input to an intensity. Visualizing this expression, Figure 8.20(a) is a two-dimensional slice of marble rendered with the colour spline given in Figure 8.20(b). Next we add turbulence:

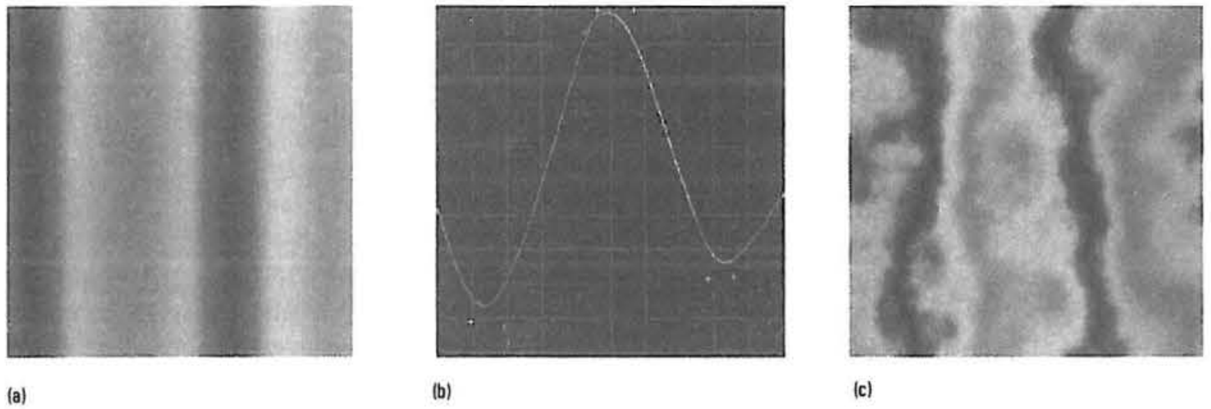


Figure 8.20
Simulating marble.
(a) Unturbulated slice obtained by using the spline shown in (b). (b) Colour spline used to produce (a). (c) Marble section obtained by turbulating the slice shown in (a).

$$\text{marble}(x) = \text{marble\_colour}\ (\sin(x + \text{turbulence}(x))$$

to give us Figure 8.20(c), a convincing simulation of marble texture. Figure 8.21 (Colour Plate) shows the effect in three dimensions.

Of course, use of the turbulence function need not be restricted to modulate just the colour of an object. Any parameter that affects the appearance of an object can be turbulated. Oppenheimer (1986) turbulates a sawtooth function to bump map the ridges of bark on a tree. Turbulence can drive the transparency of objects such as clouds. Clouds can be modelled by texturing an opacity map onto a sphere that is concentric with the earth. The opacity map can be created with a paint program; clouds are represented as white blobs with soft edges that fade into complete transparency. These edges become turbulent after perturbation of the texture coordinates. In an extension to his earlier work, Perlin (1989) uses turbulence to volumetrically render regions of space rather than just evaluating texture at the surface of an object. Solid texture is used to modulate the geometry of an object as well as its appearance. Density modulation functions that specify the soft regions of objects are turbulated and rendered using a ray marching algorithm. A variety of applications are described, including erosion, fire and fur.

8.7.3 Three-dimensional texture and animation

The turbulence function can be defined over time as well as space simply by adding an extra dimension representing time, to the noise integer lattice. So the lattice points will now be specified by the indices (i, j, k, l) enabling us to extend the parameter list to noise (x, t) and similarly for turbulence (x, t). Internal to these procedures the time axis is not treated any differently from the three spatial axes.

For example, if we want to simulate fire, the first thing that we do is to try to represent its basic form functionally, that is, a 'flame shape'. The completely ad hoc nature of this functional sculpting is apparent here. The final form decided

on was simply that which after experimentation gave the best results. We shall work in two space due to the expense of the three-dimensional volumetric approach referred to at the end of the last section.

A flame region is defined in the xy plane by the rectangle with minimax coordinates $(-b, 0)$, (b, h). Within this region the flame's colour is given by:

$$\text{flame}(x) = (1 - y/h)\ \text{flame\_colour}(\text{abs}(x/b))$$

This is shown schematically in Figure 8.22 (Colour Plate). *Flame_colour* (x) consists of three separate colour splines that map a scalar value x to a colour vector. Each of the R, G, B splines have a maximum intensity at $x = 0$ which corresponds to the centre of the flame and a fade-off to zero intensity at $x = 1$. The green and blue splines go to zero faster than the red. The colour returned by *flame_colour*() is weighted according to its height from the base of the flame to get an appropriate variation along y. The flame is rendered by applying *flame*() to colour a rectangular polygon that covers the region of the flames definition. The opacity of the polygon is also textured by using a similar functional construction. Figure 8.22 also shows the turbulated counterpart obtained by introducing the turbulence function thus:

$$\text{flame}(x, t) = (1 - y/h)\ \text{flame\_colour}(\text{abs}(x/b) + \text{turbulence}(x, t)$$

To animate the flame we simply render successive slices of noise which are perpendicular to the time axis and equispaced by an amount corresponding to the frame interval. It is as if we are translating the polygon along the time axis. However, mere translation in time is not enough, recognizable detail in the flame, though changing shape with time, remained curiously static in space. This is because there is a general sense of direction associated with a flame, convection sends detail upwards. This was simulated, and immediately gave better results, by moving the polygon down in y as well as through time, as shown in Figure 8.23. The final construction is thus:

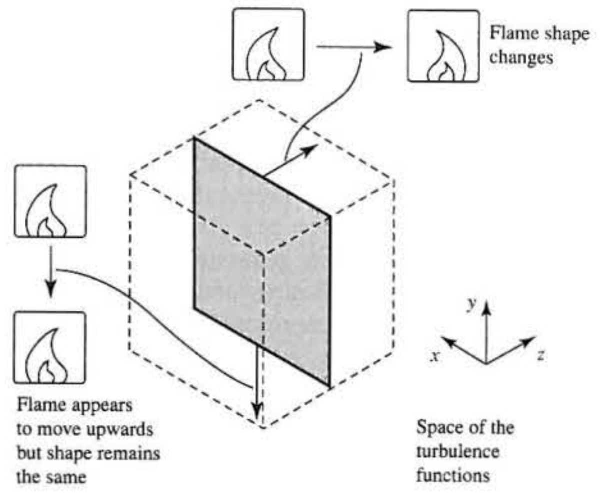

Flame shape changes
Flame appears to move upwards but shape remains the same
Space of the turbulence functions
y
x
z

Figure 8.23
Animating turbulence for a two-dimensional object.

flame(x, t)=(1 – y/h)flame_colour(abs(x/b)+turbulence(x+(0, $t\Delta y$, 0), t))

where Δy is the distance moved in y by the polygon relative to the noise per unit time.

8.7.4 Three-dimensional light maps

In principle there is no reason why we cannot have three-dimensional light maps – the practical restriction is the vast memory resources that would be required. In the event that it is possible we have a method of caching the reflected light at every point in the scene. We use any view-independent rendering method and assign the calculated light intensity at point (x, y, z) in object space to $T(x, y, z)$. It is interesting to now compare our pre-calculation mapping methods.

With environment mapping we cache all the incoming illumination at a *single* point in object space in a two-dimensional map which is labelled by the direction of the incoming light at the point. A reflected view vector is then used to retrieve the reflected light directed towards the user. These are normally used for perfect specular surfaces and give us fast view-dependent effects.

With two-dimensional light maps we cache the reflected light for each surface in the scene in a set of two-dimensional maps. Indexing into these maps during the rendering phase depends on the method that was used to sample three-dimensional object space. We use these to cache view-independent non-dynamic lighting.

With three-dimensional light maps we store reflected light at a point in a three-dimensional structure that represents object space. Three-dimensional light maps are a subset of light fields (see Chapter 16).

8.8 Anti-aliasing and texture mapping

As we have discussed in the introduction to this chapter, artefacts are extremely problematic in texture mapping and most textures produce visible artefacts unless the method is integrated with an anti-aliasing procedure. Defects are highly noticeable, particularly in texture that exhibits coherence or periodicity, as soon as the predominant spatial frequency in the texture pattern approaches the dimension of a pixel. (The classic example of this effect is shown in Figure 8.3.) Artefacts generated by texture mapping are not well handled by the common anti-aliasing method – such as supersampling – and because of this standard two-dimensional texture mapping procedures usually incorporate a specific anti-aliasing technique.

Anti-aliasing in texture mapping is difficult because, to do it properly, we need to find the pre-image of a pixel and sum weighted values of $T(u, v)$ that fall within the extent of the pre-image to get a single texture intensity for the pixel. Unfortunately the shape of the pre-image changes from pixel to pixel and this

filtering process consequently becomes expensive. Refer again to Figure 8.2. This shows that when we are considering a pixel its pre-image in texture space is, in general, a curvilinear quadrilateral, because the net effect of the texture mapping and perspective mapping is of a non-linear transformation. The figure also shows, for the diagonal band, texture for which, unless this operation is performed or approximated, erroneous results will occur. In particular, if the texture map is merely sampled at the inverse mapping of the pixel centre then the sampled intensity may be correct if the inverse image size of the pixel is sufficiently small, but in general it will be wrong.

In the context of Figure 8.24(a), anti-aliasing means approximating the integration shown in the figure. An approximate, but visually successful, method ignores the shape but not the size or extent of the pre-image and pre-calculates all the required filtering operations. This is mip-mapping invented by Williams (1983) and probably the most common anti-aliasing method developed specifically for texture mapping. His method is based on pre-calculation and an assumption that the inverse pixel image is reasonably close to a square. Figure 8.24(b) shows the pixel pre-image approximated by a square. It is this approximation that enables the anti-aliasing or filtering operation to be pre-calculated. In fact there are two problems. The first is more common and is known as compression or minification. This occurs when an object becomes small in screen space and consequently a pixel has a large pre-image in texture space. Figure 8.24(c) shows this situation. Many texture elements (sometimes called 'texels') need to be mapped into a single pixel. The other problem is called magnification. Here an object becomes very close to the viewer and only part of the object may occupy the whole of screen space, resulting in pixel pre-images that have less area than one texel (Figure 8.24(d)). Mip-mapping deals with compression and some elaboration to mip-mapping is usually required for the magnification problem.

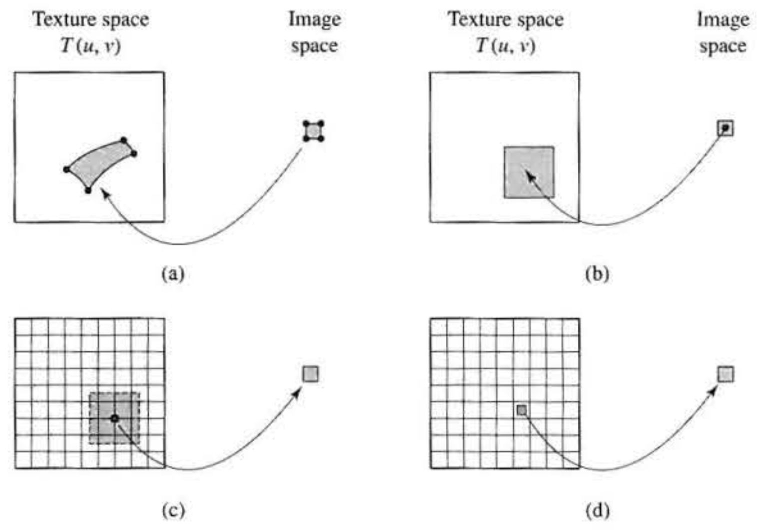


Figure 8.24
Mip-mapping approximations. (a) The pre-image of a pixel is a curvilinear quadrilateral in texture space.
(b) A pre-image can be approximated by a square.
(c) Compression is required when a pixel maps onto many texels.
(d) Magnification is required when a pixel maps onto less than one texel.

In mip-mapping, instead of a texture domain comprising a single image, Williams uses many images, all derived by averaging down the original image to successively lower resolutions. In other word they form a set of pre-filtered texture maps. Each image in the sequence is exactly half the resolution of the previous. Figure 8.25 shows an approximation to the idea. An object near to the viewer, and large in screen space, selects a single texel from a high-resolution map. The same object further away from the viewer and smaller in screen space selects a single texel from a low-resolution map. An appropriate map is selected by a parameter D. Figure 8.26 (Colour Plate) shows the mip-map used in Figure 8.8.

In a low-resolution version of the image each texel represents the average of a number of texels from the previous map. By a suitable choice of D, an image at appropriate resolution is selected and the filtering cost remains constant – the many texels to one pixel cost problem being avoided. The centre of the pixel is mapped into that map determined by D and this single value is used. In this way the original texture is filtered and, to avoid discontinuities between the images at varying resolutions, different levels are also blended. Blending between levels occurs when D is selected. The images are discontinuous in resolution but D is a continuous parameter. Linear interpolation is carried out from the two nearest levels.

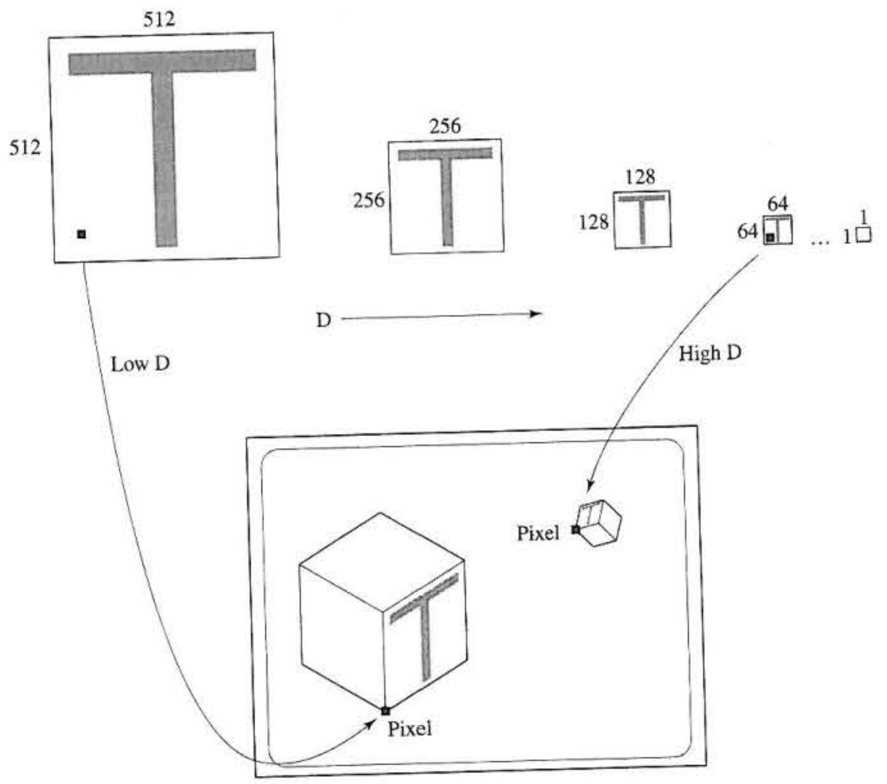


Figure 8.25
Showing the principle of mip-mapping.

Williams selects D from:

$$D = \text{max\_of}\left(\left(\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2\right)^{1/2}, \left(\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2\right)^{1/2}\right)$$

where ∂u and ∂v are the original dimensions of the pre-image in texture space and $\partial x = \partial y = 1$ for a square pixel.

A 'correct' or accurate estimation of D is important. If D is too large then the image will look blurred, too small and aliasing artefacts will still be visible. Detailed practical methods for determining depending on the mapping context are given in Watt and Watt (1992).

In a theoretical sense the magnification problem does not exist. Ideally we would like mip-maps that can be used at any level of detail, but in practice, storage limitations restrict the highest resolution mask to, say, 512 × 512 texels. This problem does not seem to have been addressed in the literature and the following two approaches are supplied by Silicon Graphics for their workstation family. Silicon Graphics suggest two solutions. First, to simply extrapolate beyond the highest resolution mip-map, and a more elaborate procedure that extracts separate texture information into low and high frequency components.

Extrapolation is defined as:

LOD(+1) = LOD(0) + (LOD(0) – LOD(–1))

where LOD (level of detail) represents mip-maps as follows:

LOD(+1) is the extrapolated mip-map
LOD(0) is the highest resolution stored mip-map
LOD(–1) is the next highest resolution stored mip-map

This operation derives an extrapolated mip-map of blocks of 4 × 4 pixels over which there is no variation. However, the magnification process preserves edges – hence the name.

Extrapolation works best when high frequency information is correlated with low frequency structural information, that is when the high frequency information represents edges in the texture. For example, consider that texture pattern is made up of block letters. Extrapolation will blur/magnify the interior of the letters, while keeping the edges sharp.

When high frequency information is not correlated with low frequency information, extrapolation causes blurring. This occurs with texture that tends to vary uniformly throughout, for example wood grain. Silicon Graphics suggest separating the low and high frequency information and converting a high resolution (unstorable at, say, 2K × 2K) into a 512 × 512 map that stores low frequency or structural information and a 256 × 256 map that stores high frequency detail. This separation can be achieved accurately using classical filtering techniques. Alternatively a space domain procedure is as follows:

(1) Make a 512 × 512 low frequency map by simply re-sampling the original 2K × 2K map.

(2) Make the 256 × 256 detail mask as follows:
   (i) Select a 256 × 256 window from the original map that contains representative high frequency texture.
   (ii) Re-sample this to 64 × 64 and re-scale to 256 × 256 resulting in a blurred version of the original 256 × 256 map.
   (iii) Subtract the blurred map from the original, adding a bias to make the subtrahend image unsigned. This results in a 256 × 256 high frequency.

Now when magnification is required a mix of the 512 × 512 low resolution texture with the high resolution detail is used.

8.9 Interactive techniques in texture mapping

One of the main problems in designing a conventional two-dimensional texture map is the visualization of the result on the rendered object. Say an artist or a designer is creating a texture map by painting directly in the two-dimensional *uv* space of the map. We know that the distortion of the map, when it is 'stuck' on the object is both a function of the shape of the object and the mapping method that is used. To design a texture interactively the artist needs to see the final rendered object and have some intuition of the mapping mechanism so that he can predict the effect of changes made to the texture map.

We will now describe two interactive techniques. In the first the designer paints in *uv* or texture space. The second attempts to make the designer think that he is painting directly on the object in 3D world space.

The first technique is extremely simple and was evolved to texture animals/objects that exhibit a plane of symmetry. It is simply an interactive version of two-part texture mapping with a plane as the intermediate object (see Section 8.1.2). The overall idea is shown in Figure 8.27. The animal model is enclosed in a bounding box. The texture map $T(u, v)$ is then 'stuck' on the two faces of the box using the 'minimax' coordinates of the box and points in $T(u, v)$ are projected onto the object using a parallel projection, with projectors normal to the plane of symmetry.

The second technique is to allow the artist to interact directly with the rendered version on the screen. The artist applies the texture using an interactive device simulating a brush and the effect on the screen is as if the painter was applying paint directly to the 3D object. It is easy to see the advantages of such a method by looking first at how it differs from a normal 2D paint program which basically enables a user to colour selected pixels on the screen.

Say we have a sphere (circle in screen space). With a normal paint program, if we selected, say, the colour green and painted the sphere, then unless we explicitly altered the colour, the sphere's projection would be filled with the selected uniform green colour. However, the idea of using a paint interaction in object space is that as you apply the green paint its colour changes according to the application of the Phong shading equation, and if the paint were gloss a specular highlight would appear. Extending the idea to texture mapping means

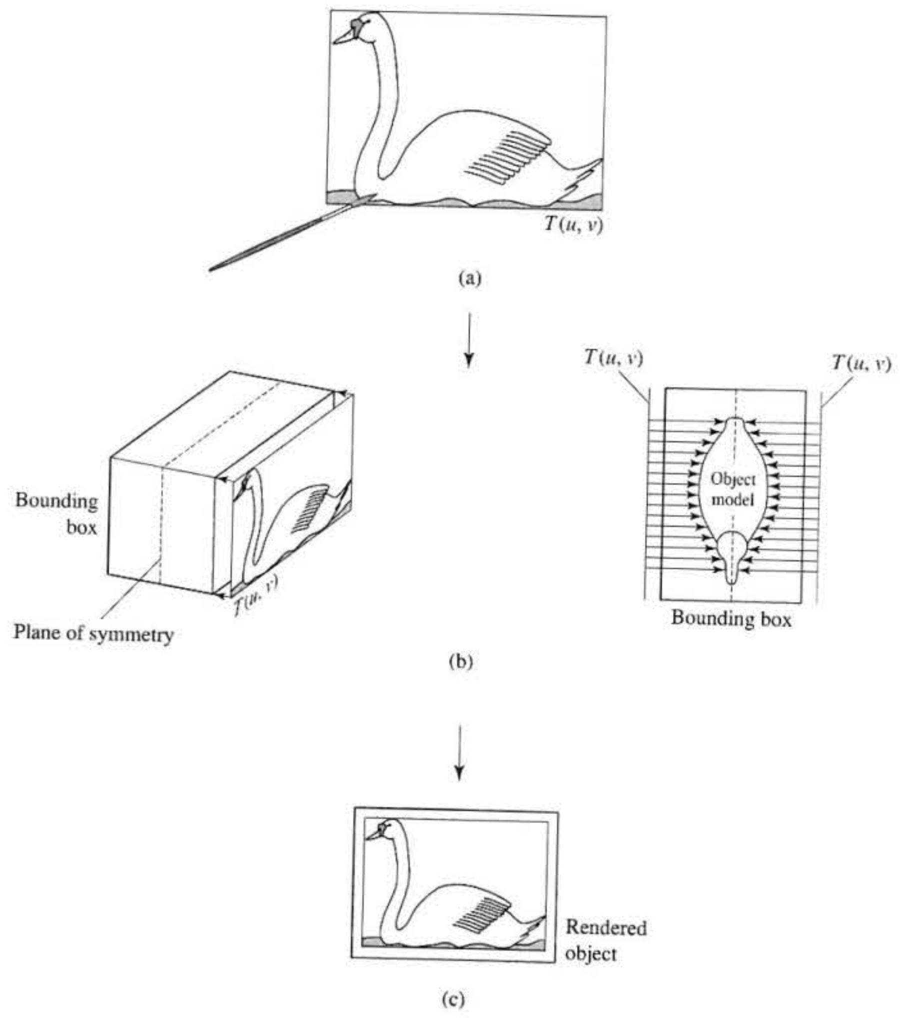


Figure 8.27 Interactive texture mapping – painting in $T(u,v)$ space. (a) Texture is painted using an interactive paint program. (b) Using the object's bounding box, the texture map points are projected onto the object. All projectors are parallel to each other and normal to the bounding box face. (c) The object is rendered, the 'distortion' visualized and the artist repeats the cycle if necessary.

that the artist can paint the texture on the object directly and the program, reversing the normal texture mapping procedure, can derive the texture map from the object. Once the process is complete, new views of the object can be rendered and texture mapped in the normal way.

This approach requires a technique that identifies, from the screen pixel that is being pointed to, the corresponding point on the object surface. In the method described by Hanrahan and Haeberli (1990) an auxiliary frame buffer, known as an item buffer, is used. Accessing this buffer with the coordinates of the screen cursor gives a pointer to the position on the object surface and the corresponding (u, v) coordinate values for the texture map. Clearly we need an object representation where the surface is everywhere parametrized and Hanrahan and Haeberli (1990) divide the object surface into a large number of micropolygons. The overall idea is illustrated in Figure 8.28.

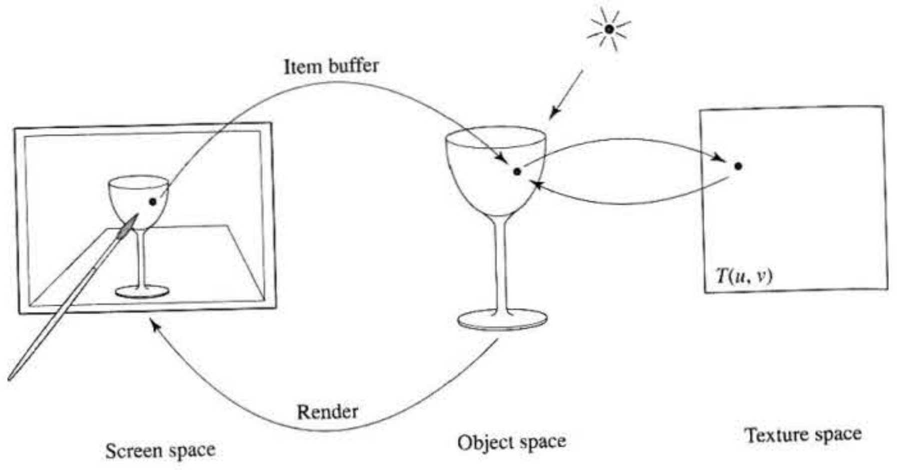


Figure 8.28
Iterative texture mapping – painting in object space.

9 Geometric shadows

Introduction

This chapter deals with the topic of 'geometric' shadows or algorithms that calculate the shape of an area in shadow but only guess at its reflected light intensity. This restriction has long been tolerated in mainstream rendering; the rationale presumably being that it is better to have a shadow with a guessed intensity than to have no shadow at all.

Shadows like texture mapping are commonly handled by using an empirical add-on algorithm. They are pasted into the scene like texture maps. The other parallel with texture maps is that the easiest algorithm to use computes a map for each light source in the scene, known as a shadow map. The map is accessed during rendering just as a texture map is referenced to find out if a pixel is in shadow or not. Like the Z-buffer algorithm in hidden surface removal, this algorithm is easy to implement and has become a pseudo-standard. Also like the Z-buffer algorithm it trades simplicity against high memory cost.

Shadows are important in scenes. A scene without shadows looks artificial. They give clues concerning the scene, consolidate spatial relationships between objects and give information on the position of the light source. To compute shadows completely we need knowledge both of their shape and the light intensity inside them. An area of the scene in shadow is not completely bereft of light. It is simply not subject to direct illumination, but receives indirect illumination from another nearby object. Thus shadow intensity can only be calculated taking this into account and this means using a global illumination model such as radiosity. In this algorithm (see Chapter 11) shadow areas are treated no differently from any other area in the scene and the shadow intensity is a light intensity, reflected from a surface, like any other.

Shadows are a function of the lighting environment. They can be hard edged or soft edged and contain both an umbra and a penumbra area. The relative size

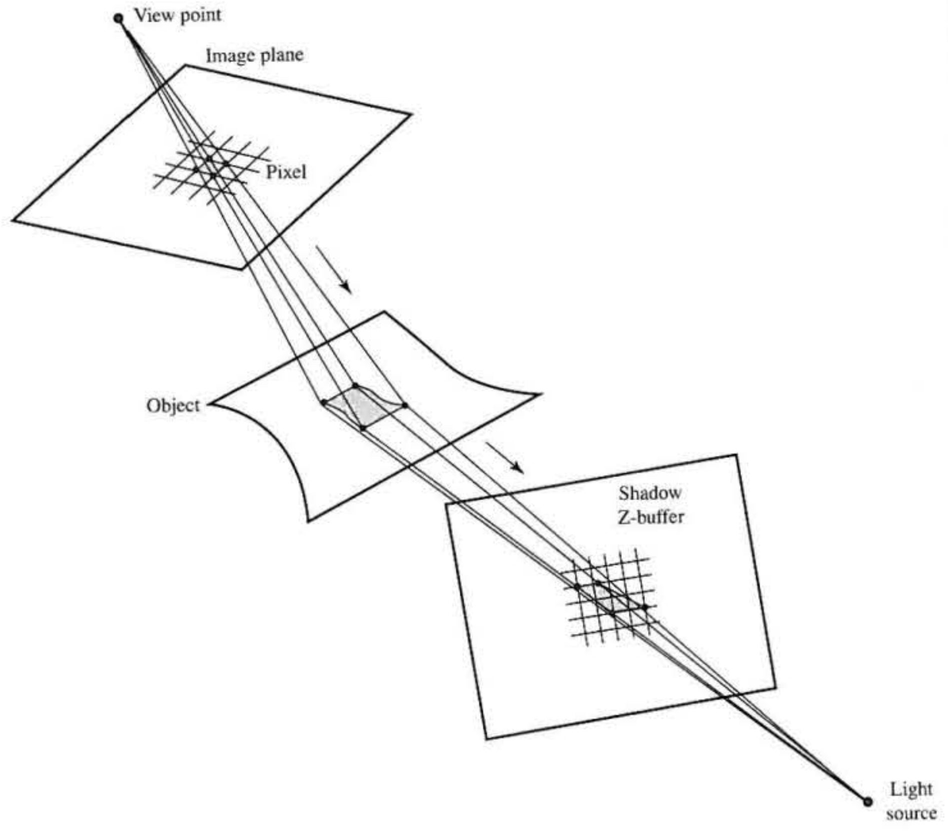


Figure 9.7
'Pre-image' of a pixel in the shadow Z-buffer.

(3) We use this fraction to give an appropriate attenuated intensity. The visual effect of this is that the hard edge of the shadow will be softened for those pixels that straddle a shadow boundary.

Full details of this approach are given in Reeves *et al.* (1987). The price paid for this anti-aliasing is a considerable increase in processing time. Pre-filtering techniques (see Chapter 14) cannot be used and a stochastic sampling scheme for integrating within the pixel pre-image in the shadow Z-buffer map is suggested in Reeves *et al.* (1987).

10 Global illumination

Introduction

In computer graphics, global illumination is the term given to models which render a view of a scene by evaluating the light reflected from a point x taking into account all illumination that arrives at a point. That is we consider not only the light arriving at the point directly from light sources but all indirect illumination that may have originated from a light source via other objects.

It is probably the case that in the general pursuit of photo-realism, most research effort has gone into solving the global illumination problem. Although, as we have seen in Chapter 7, considerable parallel work has been carried out with local reflection models, workers have been attracted to the difficult problem of simulating the interaction of light with an entire environment. Light has to be tracked through the environment from emitter(s) to sensor(s), rather than just from an emitter to a surface then directly to the sensor or eye. Such an approach does not then require add-on algorithms for shadows which are simply areas in which the illumination level is reduced due to the proximity of a nearby object. Other global illumination effects such as reflection of objects in each other and transparency effects can also be correctly modelled.

It is not clear how important global illumination is to photo-realism. Certainly it is the case that we are accustomed to 'closed' man-made environments, where there is much global interaction, but the extent to which this interaction has to be simulated, to achieve a degree of realism acceptable for most computer graphics applications, is still an open question. Rather, the problem has been vigorously pursued as a pure research problem in its own right on the assumption that improvements in the accuracy of global interaction will be valuable.

Two established (partial) global algorithms have now emerged. These are ray tracing and radiosity and, for reasons that will soon become clear, they both, in their most commonly implemented forms, simulate only a subset of global interaction: ray tracing attending to (perfect) specular interaction and radiosity to (perfect) diffuse interaction. In other words, current practical solutions to the problem deal with its inherent intractability by concentrating on particular global interactions, ignoring the remainder and by considering interactions to be perfect. In the case of specular interaction 'perfect' means that an infinitesimally thin beam hitting a surface reflects without spreading – the surface is assumed perfect. In the case of perfect diffuse interaction we assume that an incoming beam of light reflects equally in all directions into the hemisphere centred at the point of reflection.

Ignoring finite computing resources, a solution to the global interaction problem is simply stated. We start at the light source(s) and follow every light path (or ray of light) as it travels through the environment stopping when the light hits the eye point, has its energy reduced below some minimum due to absorption in the objects that it has encountered, or travels out of the environment into space. To see the relevance of global illumination algorithms we need ways of describing the problem – models that capture the essence of the behaviour of light in an environment. In this chapter we will introduce two models of global illumination and give an overview of the many and varied approaches to global illumination. We devote separate chapters to the implementation details of the two well-established methods of ray tracing and radiosity.

We should note that it is difficult to categorize global illumination algorithms because most use a combination of techniques. Is two-pass ray tracing, for example, to be considered as a global illumination method or as an extension to ray tracing? Thus the breakdown by technique that appears in this chapter inevitably contains algorithms that straddle more than one category and the sorting is simply the author's preference.

10.1 Global illumination models

We start by introducing two 'models' of the global illumination problem. The first is a mathematical formulation and the second is a classification in terms of the nature of the type of interaction that can occur when light travels from one

surface to the other. The value of such models is that they enable a comparison between the multitude of global illumination algorithms most of which evaluate a less than complete solution. By their nature the algorithms consist of a wealth of heuristic detail and the global illumination models facilitate a comparison in terms of which aspects are evaluated and which are not.

10.1.1 The rendering equation

The first model that we will look at was introduced into the computer graphics literature in 1986 by Kajiya (Kajiya 1986) and is known as the rendering equation. It encapsulates global illumination by describing what happens at a point x on a surface. It is a completely general mathematical statement of the problem and global illumination algorithms can be categorized in terms of this equation. In fact, Kajiya states that its purpose:

is to provide a unified context for viewing them [rendering algorithms] as more or less accurate approximations to the solution for a single equation.

The integral in Kajiya's original notation is given by:

$$I(x, x') = g(x, x')[\varepsilon(x, x') + \int_s \rho(x, x', x'')\, I(x', x'')\mathrm{d}x'']$$

where:

$I(x, x')$ is the transport intensity or the intensity of light passing from point x' to point x. Kajiya terms this the unoccluded two point transport intensity.

$g(x, x')$ is the visibility function between x and x'. If x and x' cannot 'see' each other then this is zero. If they are visible then g varies as the inverse square of the distance between them.

$\varepsilon(x, x')$ is the transfer emittance from x' to x and is related to the intesity of any light self-emitted by point x' in the direction of x.

$\rho(x, x', x'')$ is the scattering term with respect to direction x' and x''. It is the intensity of the energy scattered towards x by a surface point located at x' arriving from point or direction x''. Kajiya calls this the unoccluded three-point transport reflectance. It is related to the BRDF (see Chapter 7) by:

$$\rho(x, x', x'') = \rho(\theta'_{\text{in}}, \phi'_{\text{in}}, \theta'_{\text{ref}}, \phi'_{\text{ref}}) \cos\theta \cos\theta'_{\text{ref}}$$

where θ' and ϕ' are the azimuth and elevation angles related to point x' (see Section 7.3) and θ is the angle between the surface normal at point x and the line $x'x$.

The integral is over s, all points on all surfaces in the scene, or equivalently over all points on the hemisphere situated at point x'. The equation states that the transport intensity from point x' to point x is equal to (any) light emitted from x' towards x plus the light scattered from x' towards x from all other surfaces in the scene – that is, that originate from direction x''.

Expressed in the above terms the rendering equation implies that we must have:

- A model of the light emitted by a surface ε().
- A representation of the BRDF ρ() for each surface.
- A method for evaluating the visibility function.

We have already met all these factors; here the formulation gathers them into a single equation. The important general points that come out of considering the rendering equation are:

(1) The complexity of the integral means that it cannot be evaluated analytically and most practical algorithms reduce the complexity in some way. The direct evaluation of the equation can be undertaken by using Monte Carlo methods and many algorithms follow this approach.

(2) It is a view-independent statement of the problem. The point x' is every point in the scene. Global illumination algorithms are either view independent – the common example is the radiosity algorithm – or view dependent where only those points x' visible from the viewing position are evaluated. View dependence can be seen as a way in which the inherent complexity of the rendering equation is reduced. (See Section 10.8 for a more detailed discussion on view dependence/independence.)

(3) It is a recursive equation – to evaluate $I(x, x')$ we need to evaluate $I(x', x'')$ which itself will use the same equation. This gives rise to one of the most popular practical methods for solving the problem which is to trace light from the image plane, in the reverse direction of light propagation, following a path that reflects from object to object. Algorithms that adopt this approach are: path tracing, ray tracing and distributed ray tracing, all of which will be described later.

10.1.2 Radiance, irradiance and the radiance equation

The original form of the rendering equation is not particularly useful in global illumination methods and in this section we will introduce definitions that enable us to write it in a different form called the radiance equation.

Radiance L is the fundamental radiometric quantity and for a point in three-dimensional space it is the light energy density measured in W/(sr–m^2). The radiance at a point is a function of direction and we can define a radiance distribution function for a point. This will generally be discontinuous as the two-dimensional example in Figure 10.1 demonstrates. Such a distribution function exists at all points in three-dimensional space and radiance is therefore a five-dimensional quantity. Irradiance is the integration of incoming radiance over all directions:

$$E = \int_{\Omega} L_{\text{in}} \cos\theta \, d\omega$$

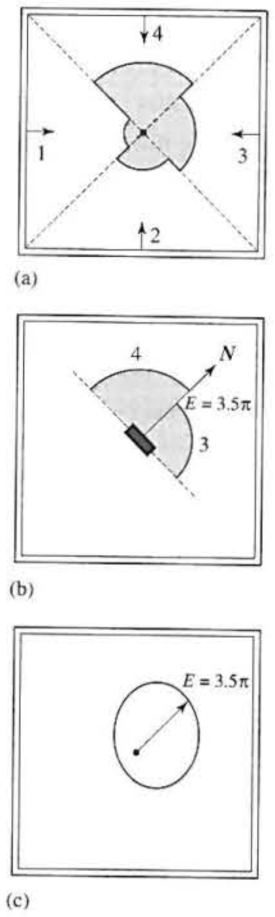


Figure 10.1
Radiance, irradiance and irradiance distribution function (after Greger *et al.* (1998)).

(a) A two-dimensional radiance distribution for a point in the centre of a room where each wall exhibits a different radiance.

(b) The field radiance for a point on a surface element. Irradiance E is the cosine weighted average of the radiance – in this case 3.5 π.

(c) If the surface element is rotated an irradiance distribution function is defined.

where:

L_{in} is the incoming or field radiance from direction ω
θ is the angle between the surface normal and ω

If L_{in} is constant, we have for a diffuse surface:

$$L_{\text{diffuse}} = \rho E/\pi$$

The distinction between these two quantities is important in global illumination algorithms as the form of the algorithm can be classified as 'shooting' or 'gathering'. Shooting means distributing radiance from a surface and gathering means integrating the irradiance or accumulating light flux at the surface. (Radiosity B is closely related to irradiance having units W/m^2.)

An important practical point concerning radiance and irradiance distribution functions is that while the former is generally discontinuous the latter is generally continuous, except for shadow boundaries. This is demonstrated in Figure 10.1 which shows that in this simple example the irradiance distribution function will be continuous because of the averaging effect of the integration.

The rendering equation can be recast as the radiance equation which in its simplest form is:

$$L_{\text{ref}} = \int \rho L_{\text{in}}$$

Including the directional dependence, we then write:

$$L_{\text{ref}}(\mathbf{x}, \omega_{\text{ref}}) = L_e(\mathbf{x}, \omega_{\text{ref}}) + \int_{\Omega} \rho(\mathbf{x}, \omega_{\text{in}} \to \omega_{\text{out}}) L_{\text{in}}(\mathbf{x}, \omega_{\text{in}}) \cos\theta_{\text{in}}\, d\omega_{\text{in}}$$

where the symbols are defined in Figure 10.2(b). This can be modified so that the integration is performed over all surfaces – usually more convenient in practical algorithms – rather than all incoming angles and this gives the rendering equation in terms of radiance:

$$L_{\text{ref}}(\mathbf{x}, \omega_{\text{ref}}) = L_e(\mathbf{x}, \omega_{\text{ref}}) + \int_{S} \rho(\mathbf{x}, \omega_{\text{in}} \to \omega_{\text{out}}) L_{\text{in}}(\mathbf{x}', \omega_{\text{in}}) g(\mathbf{x}, \mathbf{x}') \cos\theta_{\text{in}} \frac{\cos\theta_0\, dA}{\|\mathbf{x} - \mathbf{x}'\|^2}$$

which now includes the visibility function. This comes about by expressing the solid angle $d\omega_{\text{in}}$ in terms of the projected area of the differential surface region visible in the direction of ω_{in} (Figure 10.2(c)):

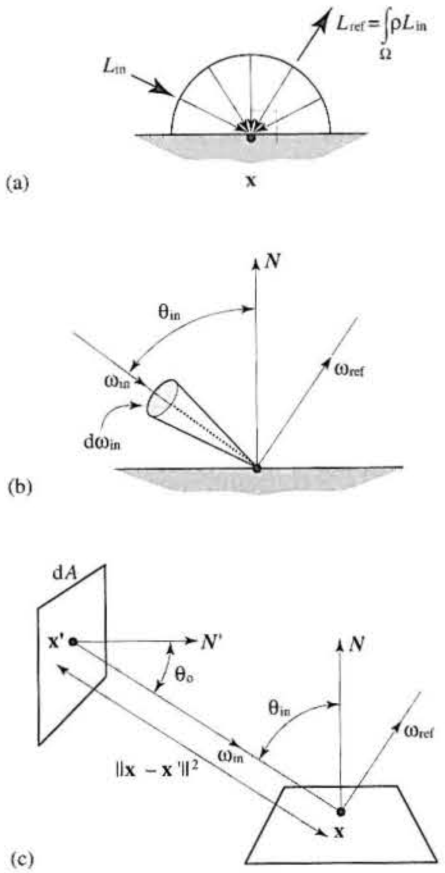


Figure 10.2
The radiance equation.

(a) The domain of integration is the hemisphere of all incoming directions.

(b) Symbols used to define the directional dependence.

(c) $\cos\theta_0\, dA/\|\mathbf{x} - \mathbf{x}'\|^2$ is the projected area of dA visible in the direction ω_{in}.

$$d\omega_{\text{in}} = \frac{\cos\theta_0\, dA}{\|\mathbf{x} - \mathbf{x}'\|^2}$$

10.1.3 Path notation

Another way of categorizing the behaviour of global illumination algorithms is to detail which surface-to-surface interactions that they implement or simulate. This is a much simpler non-mathematical categorization and it enables an easy comparison and classification of the common algorithms. We consider which interactions between pairs of interacting surfaces are implemented as light travels from source to sensor. Thus at a point, incoming light may be scattered or reflected diffusely or specularly and may itself have originated from a specular or diffuse reflection at the previous surface in the path. We can then say that for pairs of consecutive surfaces along a light path we have (Figure 10.3):

- Diffuse to diffuse transfer.
- Specular to diffuse transfer.
- Diffuse to specular transfer.
- Specular to specular transfer.

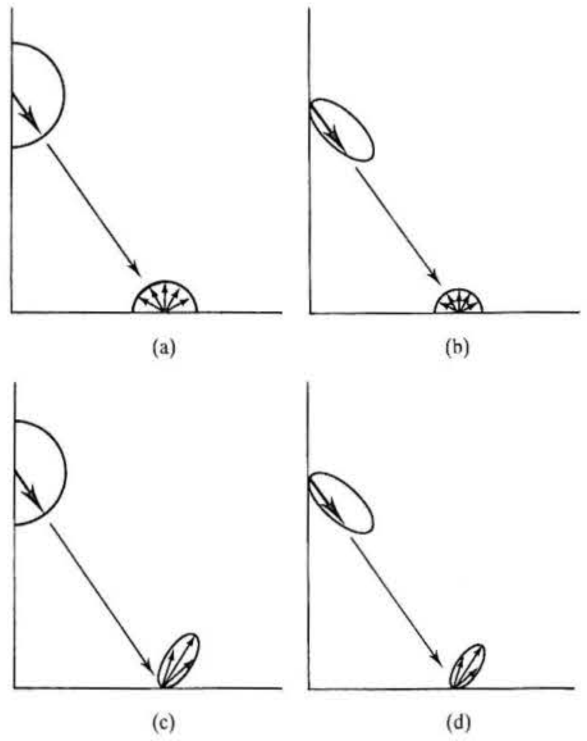


Figure 10.3
The four 'mechanisms' of light transport: (a) diffuse to diffuse; (b) specular to diffuse; (c) diffuse to specular; (d) specular to specular (after Wallace *et al.* (1987)).

In an environment where only diffuse surfaces exist only diffuse–diffuse interaction is possible and such scenes are solved using the radiosity method. Similarly an environment containing only specular surfaces can only exhibit specular interaction and (Whitted) ray tracing deals with these. Basic radiosity does not admit any other transfer mechanism except diffuse–diffuse and it excludes the important specular–specular transfer. Ray tracing, on the other hand can only deal with specular–specular interaction. More recent algorithms, such as 'backwards' ray tracing and enhancements of radiosity for specular interaction require a categorization of all the interactions in a light journey from source to sensor, and this led to Heckbert's string notation (Heckbert 1990) for listing all the interactions that occur along a path of a light ray as it travels from source (L) to the eye (E). Here a light path from the light source to the first hit is termed L, subsequent paths involving transfer mechanisms at a surface point are categorized as DD, SD, DS or SS. Figure 10.4 (also a Colour Plate) shows an example of a simple scene and various paths. The path that finally terminates in the eye is called E. The paths in the example are:

(1) LDDE For this path the viewer sees the shadow cast by the table. The light reflects diffusely from the right-hand wall onto the floor. Note that any light

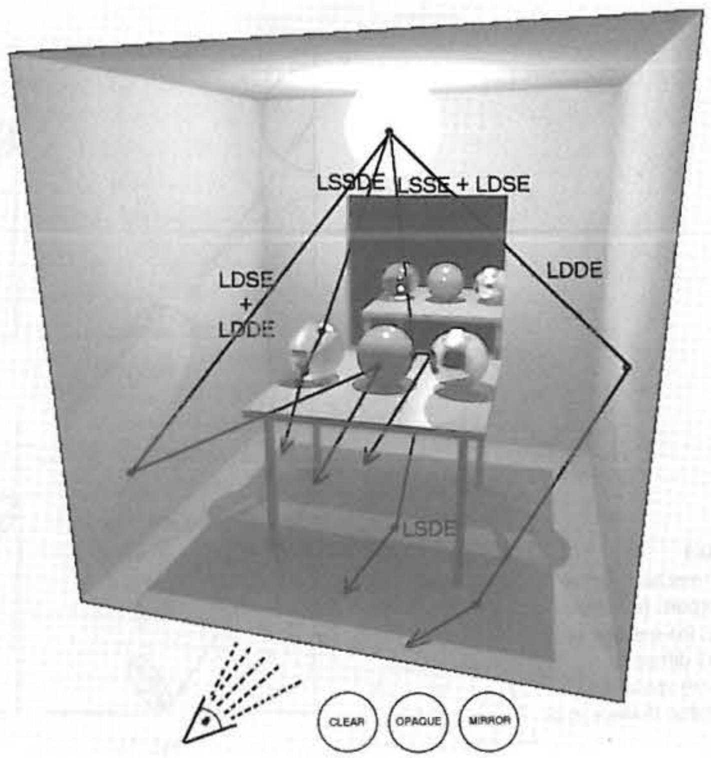


Figure 10.4
A selection of global illuminations paths in a simple environment. See also the Colour Plate version of this figure.

reflected from a shadow area must have a minimum of two interactions between L and E.

(2) LDSE + LDDE Here the user sees the dark side of the sphere which is not receiving any direct light. The light is modelled as a point source, so any area below the 'equator' of the sphere will be in shadow. The diffuse illumination reflected diffusely from the wall is directed towards the eye and because the sphere is shiny the reflection to the eye is both specular and diffuse.

(3) LSSE + LDSE Light is reflected from the perfect mirror surface to the eye and the viewer sees a reflection of the opaque or coloured ball in the mirror surface.

(4) LSDE Here the viewer sees a shadow area that is lighter than the main table shadow. This is due to the extra light reflected from the mirror and directed underneath the table.

(5) LSSDE This path has three interactions between L and E and the user sees a caustic on the table top which is a diffuse surface. The first specular interaction takes place at the top surface of the sphere and light from the point source is refracted through the sphere. There is a second specular interaction when the light emerges from the sphere and hits the diffuse table surface. The effect of the reflection is to concentrate light rays travelling through the sphere into a smaller area on the table top than they would occupy if the transparent sphere was not present. Thus the user sees a bright area on the diffuse surface.

A complete global illumination algorithm would have to include any light path which can be written as L(D|S)*E, where | means 'or' and * indicates repetition. The application of a local reflection model implies paths of type LD|S (the intensity of each being calculated separately then combined as in the Phong reflection model) and the addition of a hidden surface removal algorithm implies simulation of types LD|SE. Thus local reflection models only simulate strings of length unity (between L and E) and viewing a point in shadow implies a string which is at least of length 2.

10.2 The evolution of global illumination algorithms

We will now look at the development of popular or established global illumination algorithms using as a basis for our discussion the preceding concepts. The order in which the algorithms are discussed is somewhat arbitrary; but goes from incomplete solutions (ray tracing and radiosity) to general solutions. The idea of this section is to give a view of the algorithms in terms of global interaction.

Return to consideration of the brute force solution to the problem. There we considered the notion of starting at a light source and following every ray of light that was emitted through the scene and stated that this was a computationally intractable problem. Approximations to a solution come from constraining the light-object interaction in some way and/or only considering a small subset of the rays that start at the light and bounce around the scene. The main approximations which led to ray tracing and radiosity constrained the scene to contain only specular reflectors or only (perfect) diffuse reflectors respectively.

In what follows we give a review of ray tracing and radiosity sufficient for comparison with the other methods we describe, leaving the implementation details of these important methods for separate chapters.

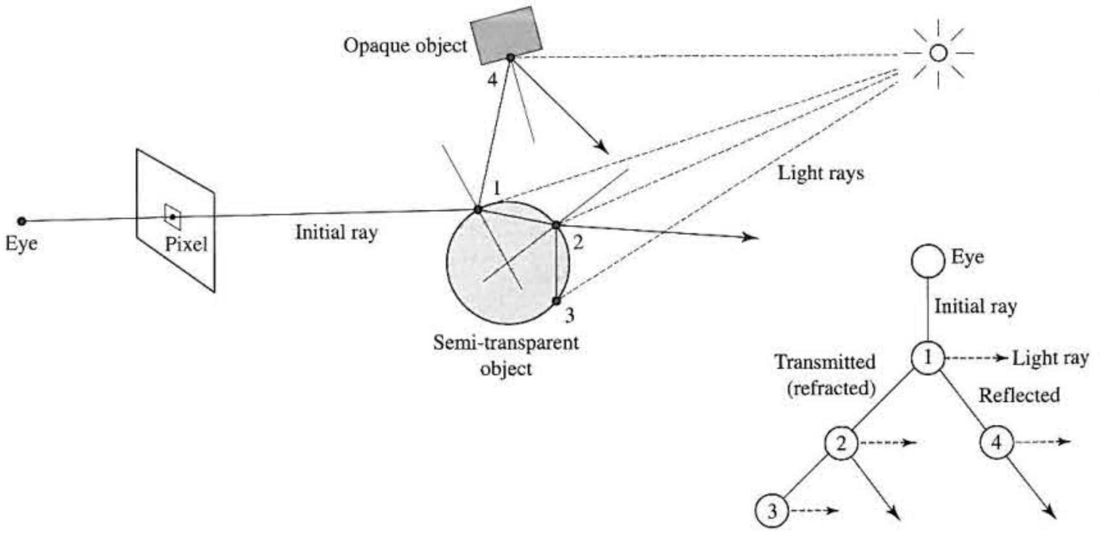
10.3 Established algorithms – ray tracing and radiosity

10.3.1 Whitted ray tracing

Whitted ray tracing (visibility tracing, eye tracing) traces light rays in the reverse direction of propagation from the eye back into the scene towards the light source. To generate a two-dimensional image plane projection of a scene using ray tracing we are only interested in these light rays that end at the sensor or eye point and therefore it makes sense to start at the eye and trace rays out into the scene. It is thus a view-dependent algorithm. A simple representation of the algorithm is shown in Figure 10.5. The process is often visualized as a tree where each node is a surface hit point. At each node we spawn a light ray and a reflected ray or a transmitted (refracted) ray or both.

Whitted ray tracing is a hybrid – a global illumination model onto which is added a local model. Consider the global interaction. The classic algorithm only includes perfect specular interaction. Rays are shot into the scene and when they hit a surface a reflected (and transmitted) ray is spawned at the point of intersection and they themselves are then followed recursively. The process stops when

Figure 10.5
Whitted ray tracing.

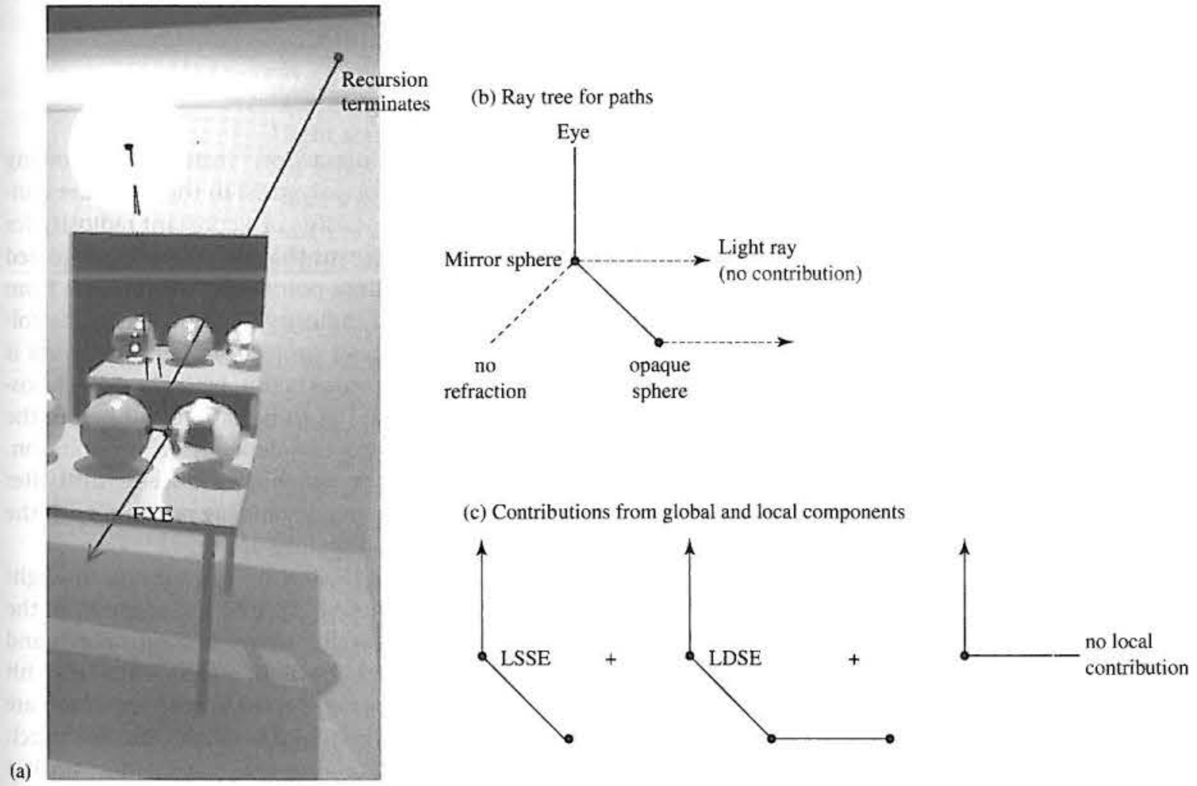


the energy of a ray drops below a predetermined minimum or if it leaves the scene and travels out into empty space or if a ray hits a surface that is perfectly diffuse. Thus the global part of ray tracing only accounts for pure specular–specular interaction. Theoretically there is nothing to stop us calculating diffuse global interaction, it is just that at every hit point an incoming ray would have to spawn reflected rays in every direction into a hemispherical surface centred on the point.

To the global specular component is added a direct contribution calculated by shooting a ray from the point to the light source which is always a point source in this model. The visibility of the point from the light source and its direction can be used to calculate a local or direct diffuse component – the ray is just L in a local reflection model. Thus (direct) diffuse reflection (but not diffuse–diffuse) interaction is considered. This is sometimes called the shadow ray or shadow feeler because if it hits any object between the point under consideration and the light source then we know that the point is shadow. However, a better term is light ray to emphasize that it is used to calculate a direct contribution (using a local reflection model) which is then passed up the tree. The main problem with Whitted ray tracing is its restriction to specular interaction – most practical scenes consist of predominantly diffuse surfaces.

Consider the LSSE + LDSE path in Figure 10.4, reproduced in Figure 10.6 together with the ray tree. The initial ray from the eye hits the perfect mirror

Figure 10.6
Whitted ray tracing: the relationship between light paths and local and global contributions for one of the cases shown in Figure 10.4.



sphere. For this sphere there is no contribution from a local diffuse model. At the next intersection we hit the opaque sphere and trace a global specular component which hits the ceiling, a perfect diffuse surface, and the recursion is terminated. Also at that point we have a contribution from the local diffuse model for the sphere and the viewer sees in the pixel associated with that ray the colour of the reflected image of the opaque sphere in the mirror sphere.

A little thought will reveal that the paths which can be simulated by Whitted ray tracing are constrained to be LS*E and LDS*E. Ray traced images therefore exhibit reflections in the surfaces of shiny objects of nearby objects. If the objects are transparent any objects that the viewer can see behind the transparent object are refracted. Also, as will be seen in Chapter 12, shadows are calculated as part of the model – but only 'perfect' or hard-edged shadows.

Considering Whitted ray tracing in terms of the rendering equation the following holds. The scattering term ρ is reduced to the law for perfect reflection (and refraction). Thus the integral over all S – the entire scene – reduces to calculating (for reflection) a single outgoing ray plus the light ray which gives the diffuse component and adding these two contributions together. Thus the recursive structure of the rendering equation is reflected perfectly in the algorithm but the integral operation is reduced to a sum of three analytically calculated components – the contributions from the reflected, transmitted and light rays.

10.3.2 Radiosity

Classic radiosity implements diffuse–diffuse interaction. Instead of following individual rays 'interaction' between patches (or polygons) in the scene are considered. The solution is view independent and consists of a constant radiosity for every patch in the scene. View independence means that a solution is calculated for every point in the scene rather than just those points that can be seen from the eye (view dependent). This implies that a radiosity solution has to be followed by another process or pass that computes a projection, but most work is carried out in the radiosity pass. A problem or contradiction with classical radiosity is that the initial discretization of the scene has to be carried out before the process is started but the best way of performing this depends on the solution. In other words, we do not know the best way to divide up the scene until after we have a solution or a partial solution. This is an outstanding problem with the radiosity method and accounts for most of its difficulty of use.

A way of visualizing the radiosity process is to start by considering the light source as an (array of) emitting patches. We shoot light into the scene from the source(s) and consider the diffuse–diffuse interaction between a light patch and all the receiving patches that are visible from the light patch – the first hit patches. An amount of light is deposited or cached on these patches which are then ordered according to the amount of energy that has fallen onto the patch and has yet to be shot back into the scene. The one with the highest unshot

energy is selected and this is considered as the next shooting patch. The process continues iteratively until a (high) percentage of the initial light energy is distributed around the scene. At any stage in the process some of the distributed energy will arrive back on patches that have already been considered and this is why the process is iterative. The process will eventually converge because the reflectivity coefficient associated with each patch is, by definition, less than unity and at each phase in the iteration more and more of the initial light is absorbed. Figure 10.7 (Colour Plate) shows a solution in progress using this algorithm. The stage shown is the state of the solution after 20 iterations. The four illustrations are:

(1) The radiosity solution as output from the iteration process. Each patch is allocated a constant radiosity.
(2) The previous solution after it has been subject to an interpolation process.
(3) The same solution with the addition of an ambient term. The ambient 'lift' is distributed evenly amongst all patches in the scene, to give an early well lit solution (this enhancement is described in detail in Chapter 11).
(4) The difference between the previous two images. This gives a visual indication of the energy that had to be added to account for the unshot radiosity.

The transfer of light between any two patches – the diffuse–diffuse interaction – is calculated by considering the geometric relationship between the patches (expressed as the form factor). Compared to ray tracing we follow light from the light source through the scene as patch-to-patch diffuse interaction, but instead of following individual rays of light, the form factor between two patches averages the effect of the paths that join the patches together. This way of considering the radiosity method is, in fact, implemented as an algorithm structure. It is called the progressive refinement method.

This simple concept has to be modified by a visibility process (not to be confused by the subsequent calculation of a projection which includes, in the normal way, hidden surface removal) that takes into account the fact that in general a patch may be only partially visible to another because of some intervening patch. The end result is the assignment of a constant radiosity to each patch in the scene – a view-independent solution which is then injected into a Gouraud-style renderer to produce a projection. In terms of path classification, conventional radiosity is LD*E.

The obvious problem with radiosity is that although man-made scenes usually consist mostly of diffuse surfaces, specular objects are not unusual and these cannot be handled by a radiosity renderer. A more subtle problem is that the scene has to be discretized into patches or polygons before the radiosities are computed and difficulties occur if this polygonization is too coarse.

We now consider radiosity in terms of the rendering equation. Radiosity is the energy per unit time per unit area and since we are only considering diffuse illumination we can rewrite the rendering equation as:

$$B(x') = \varepsilon(x') + \rho(x') \int_S B(x)F(x, x')\mathrm{d}x$$

where now the only directional dependence is incorporated in the form factor F. The equation now states that the radiosity of a surface element x is equal to the emittance term plus the radiosity radiated by all other elements in the scene onto x. The form factor F is a coefficient that is a function only of the spatial relationship between x and x' and this determines that fraction of $B(x')$ arriving at x. F also includes a visibility calculation.

10.4 Monte Carlo techniques in global illumination

In this section we will give an intuitive introduction to Monte Carlo techniques. The mathematical details are outside the intended scope of this text (see Glassner (1995) for a comprehensive treatment of Monte Carlo theory and its application to global illumination) and it is the case that the methods that use Monte Carlo techniques can be explained in algorithmic terms. However, some intuition concerning the underlying factors is necessary to appreciate the particular strategies employed by the examples which we will describe. Without this intuition it is, for example, difficult to appreciate the difference between Whitted ray tracing and Kajiya's Monte Carlo approach which he termed path tracing (Section 10.5).

Monte Carlo techniques are used to solve integrals like the rendering equation which have no analytical or numerical solution. They do this by computing the average of random samples of the integrand, adding these together and taking the average. The visual effect of this process in the final rendered image is noise. The attraction of Monte Carlo techniques is that they are easy to implement because they are conceptually simple. An equally important advantage is their generality. No *a priori* simplifications have to be made (like perfect reflectors in Whitted ray tracing and perfect diffusers in radiosity). This comes about because they point sample both the geometry of the scene and the optical properties of the surface. The problem with Monte Carlo methods comes in devising techniques where an accurate or low variance estimate of the integral can be obtained quickly.

The underlying idea of Monte Carlo methods for estimating integrals can be demonstrated using a simple one-dimensional example. Consider estimating the integral:

$$I = \int_0^1 f(x)\,\mathrm{d}x$$

I can be estimated by taking a random number $\xi \in [0,1]$ and evaluating $f(\xi)$. This is called a primary estimator. We can define the variance of the estimate as:

$$\sigma^2_{\text{prim}} = \int_0^1 f^2(x)\mathrm{d}x - f^2(\xi)$$

which for a single sample we would expect to be high. In practice we would take N samples to give a so-called secondary estimate and it is easily shown that:

$$\sigma^2_{\text{sec}} = \frac{\sigma^2_{\text{prim}}}{N}$$

This observation, that the error in the estimate is inversely proportional to the square root of the number of samples, is extremely important in practice. To halve the error, for example, we must take four times as many samples. Equivalently we can say that each additional sample has less and less effect on the result and this has to be set against the fact that computer graphics implementations tend to involve an equal, and generally high cost, per sample. Thus the main goal in Monte Carlo methods is to get the best result possible with a given number of samples N. This means strategies that result in variance reduction. The two common strategies for selecting samples are stratified sampling and importance sampling.

The simplest form of stratified sampling divides the domain of the integration into equal strata and estimates each partial integral by one or more random samples (Figure 10.8). In this way each sub-domain is allocated the same number of samples. Thus:

$$I = \int_0^1 f(x)\,\mathrm{d}x$$

$$= \sum_{i=1}^{N} \int_{S_i} f(x)\mathrm{d}x$$

$$= \frac{1}{N}\sum_{i=1}^{N} f(\xi)$$

This estimate results in a variance that is guaranteed to be lower than that obtained by distributing random samples over the integration domain. The most familiar example of stratified sampling in computer graphics is jittering in pixel sampling. Here a pixel represents the domain of the integral which is subdivided into equal strata and a sample point generated by jittering the centre point of each stratum (Figure 10.9).

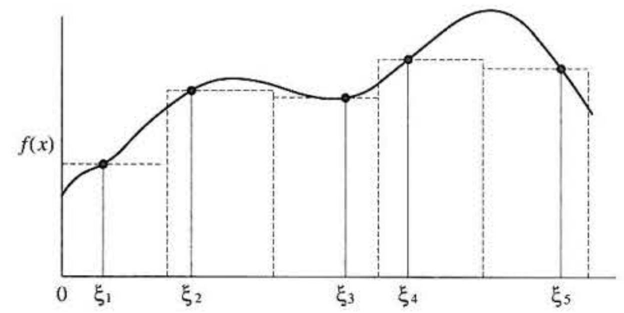


Figure 10.8
Stratified sampling of $f(x)$.

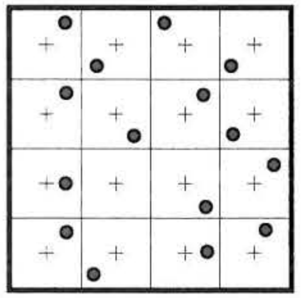


Figure 10.9
Stratified sampling in computer graphics: a pixel is divided into 16 sub-pixels and 16 sample points are generated by jittering the centre point of each sub-pixel or stratum.

As the name implies, importance sampling tends to select samples in important regions of the integrand. Importance sampling implies prior knowledge of the function that we are going to estimate an integral for, which at first sight appears to be a contradiction. However, most rendering problems involve an integrand which is the product of two functions, one of which is known *a priori* as in the rendering equation. For example, in a Monte Carlo approach to ray tracing a specular surface we would choose reflected rays which tended to cluster around the specular reflection direction thus sampling the (known) BRDF in regions where it is likely to return a high value. Thus, in general, we distribute the samples so that their density is highest in the regions where the function has a high value or where it varies significantly and quickly. Considering again our simple one-dimensional example we can write:

$$I = \int_0^1 p(x)\,\frac{f(x)}{p(x)}\,\mathrm{d}x$$

where the first term $p(x)$ is an importance weighting function. This function $p(x)$ is then the probability density function (PDF) of the samples. That is the samples need to be chosen such that they conform to $p(x)$. To do this we define $P(x)$ to be the cumulative function of the PDF:

$$P(x) = \int_0^x p(t)\mathrm{d}t$$

and choose a uniform random sample τ and evaluate $\xi = P^{-1}(\tau)$. Using this method the variance becomes:

$$\sigma^2_{\text{imp}} = \int_0^1 \left[\frac{f(x)}{p(x)}\right]^2 p(x)\mathrm{d}x - I^2$$

$$= \int_0^1 \frac{f^2(x)}{p(x)}\,\mathrm{d}x - I^2$$

The question is how do we choose $p(x)$. This can be a function that satisfies the following conditions:

$p(x) > 0$

$\int p(x)\mathrm{d}x = 1$

$P^{-1}(x)$ is computable

For example, we could choose $p(x)$ to be the normalized absolute value of $f(x)$ or alternatively a smoothed or approximate version of $f(x)$ (Figure 10.10). Any function $f(x)$ that satisfies the above conditions will not necessarily suffice. If we choose an $f(x)$ that is too far from the ideal then the efficiency of this technique will simply drop below that of a naive method that uses random samples. Importance sampling is of critical importance in global illumination algorithms that utilize Monte Carlo approaches for the simple and obvious reason that although the rendering equation describes the global illumination at each and every point in the scene we do not require a solution that is equally accurate. We require, for example, a more accurate result for a brightly illuminated specular surface than for a dimly lit diffuse wall. Importance sampling enables us to build algorithms where the cost is distributed according to the final accuracy that we require as a function of light level and surface type.

An important practical implication of Monte Carlo methods in computer graphics is that they produce stochastic noise. For example, consider Whitted ray tracing and Monte Carlo approaches to ray tracing. In Whitted ray tracing the perfect specular direction is always chosen and in a sense the integration is reduced to a deterministic algorithm which produces a noiseless image. A crude Monte Carlo approach that imitated Whitted ray tracing would produce an image where the final pixels' estimates were, in general, slightly different from the Whitted solution. These differences manifest themselves as noticeable noise. Also note that in Whitted ray tracing if we ignore potential aliasing problems we need only initiate one ray per pixel. With a Monte Carlo approach we are using samples of the rendering equation to compute an estimate of intensity of a pixel and we need to fire many rays/pixels which bounce around the scene. In Kajiya's pioneering algorithm (Kajiya 1986), described in the next section, he used a total of 40 rays per pixel.

Global illumination algorithms that use a Monte Carlo approach are all based on these simple ideas. Their inherent complexity derives from the fact that the integration is now multi-dimensional.

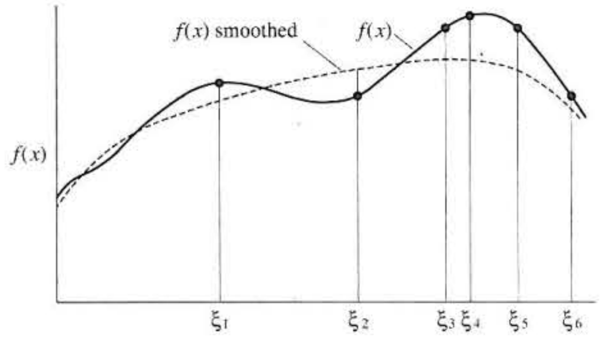


Figure 10.10
Illustrating the idea of importance sample.

10.5 Path tracing

In his classic paper that introduced the rendering equation, Kajiya (1986) was the first to recognize that Whitted ray tracing is a deterministic solution to the rendering equation. In the same paper he also suggested a non-deterministic variation of Whitted ray tracing – a Monte Carlo method that he called path tracing.

Kajiya gives a direct mathematical link between the rendering equation and the path tracing algorithm by rewriting the equation as:

$$I = g\varepsilon + gMI$$

where M is the linear operator given by the integral in the rendering equation. This can then be written as an infinite series known as a Neuman series as:

$$I = g\varepsilon + gMg\varepsilon + g(Mg)^2\varepsilon + g(Mg)^3\varepsilon + \ldots$$

where I is now the sum of a direct term, a once scattered term, a twice scattered term, etc. This leads directly to path tracing, which is theoretically known as a random walk. Light rays are traced backwards (as in Whitted ray tracing) from pixels and bounce around the scene from the first hit point, to the second, to the third, etc. The random walk has to terminate after a certain number of steps – equivalent to truncating the above series at some point when we can be sure that no further significant contributions will be encountered.

Like Whitted ray tracing, path tracing is a view-dependent solution. Previously we have said that there is no theoretical bar to extending ray tracing to handle all light–surface interactions including diffuse reflection and transmission from a hit point; just the impossibility of the computation. Path tracing implements diffuse interaction by initiating a large number of rays at each pixel (instead of, usually, one with Whitted ray tracing) and follows a single path for each ray through the scene rather than allowing a ray to spawn multiple reflected children at each hit point. The idea is shown in Figure 10.11 which can be compared with Figure 10.5. All surfaces, whether diffuse or specular can spawn a reflection/transmission ray and this contrasts with Whitted ray tracing where the encounter with a diffuse surface terminates the recursion. The other important difference is that a number of rays (40 in the original example) are initiated for each pixel enabling BRDFs to be sampled. Thus the method simulates full L(D|S)*E interaction.

A basic path tracing algorithm using a single path from source to termination will be expensive. If the random walks do not terminate on a light source then they return zero contribution to the final estimate and unless the light sources are large, paths will tend to terminate before they reach light sources. Kajiya addressed this problem by introducing a light or shadow ray that is shot towards a point on an (area) light source from each hit point in the random walk and accumulating this contribution at each point in the path (if the reflection ray from the same point directly hits the light source then the direct contribution is ignored).

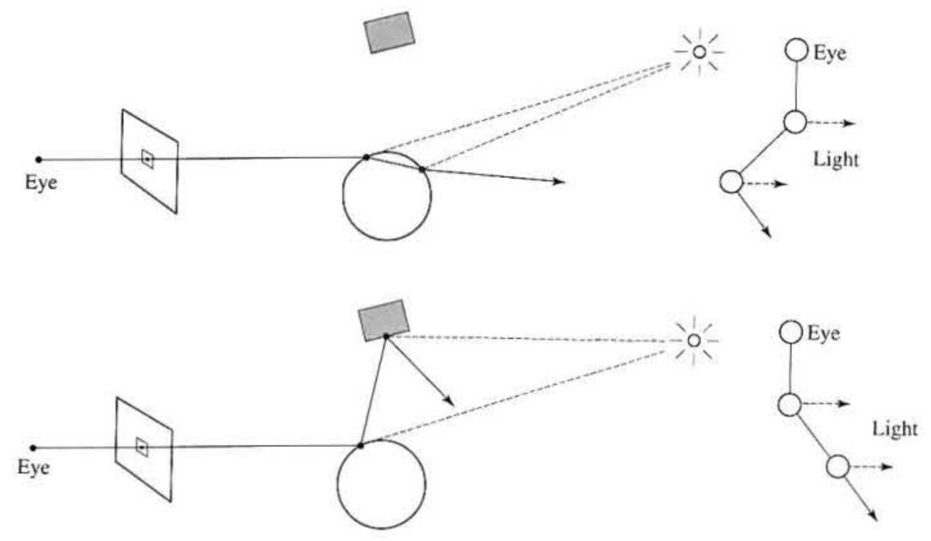


Figure 10.11
Two rays in path tracing (initiated at the same pixel).

Kajiya points out that Whitted ray tracing is wasteful in the sense that as the algorithm goes deeper into the tree it does more and more work. At the same time the contribution to the pixel intensity from events deep in the tree becomes less and less. In Kajiya's approach the tree has a branching ratio of one, and at each hit point a random variable, from a distribution based on the specular and diffuse BRDFs, is used to shoot a single ray. Kajiya points out that this process has to maintain the correct proportion of reflection, refraction and shadow rays for each pixel.

In terms of Monte Carlo theory the original algorithm reduces the variance for direct illumination but indirect illumination exhibits high variance. This is particularly true for LS*DS* E paths (see Section 10.7 for further consideration of this type of path) where a diffuse surface is receiving light from an emitter via a number of specular paths. Thus the algorithm takes a very long time to produce a good quality image. (Kajiya quotes a time of 20 hours for a 512×512 pixel image with 40 paths per pixel.)

Importance sampling can be introduced into path/ray tracing algorithms by basing it on the BRDF and ensuring that more rays are sent in directions that will return large contributions. However, this can only be done approximately because the associated PDF cannot be integrated and inverted. Another problem is that the BRDF is only one component of the integrand local to the current surface point – we have no knowledge of the light incident on this point from all directions over the hemispherical space – the field radiance (apart from the light due to direct illumination). In conventional path/ray tracing approaches all rays are traced independently of each other, accumulated and averaged into a pixel. No use is made of information gained while the process proceeds. This important observation has led to schemes that cache the information obtained during the ray trace. The most familiar of these is described in Section 10.9.

10.6 Distributed ray tracing

Like path tracing, distributed ray tracing can be seen as an extension of Whitted ray tracing or as a Monte Carlo strategy. Distributed ray tracing (distribution ray tracing, stochastic ray tracing), developed by Cook in 1986 (Cook 1986), was presumably motivated by the need to deal with the fact that Whitted ray tracing could only account for perfect specular interaction which would only occur in scenes made up of objects that consisted of perfect mirror surfaces or perfect transmitters. The effect that a Whitted ray tracer produces for (perfect) solid glass is particularly disconcerting or unrealistic. For example, consider a sphere of perfect glass. The viewer sees a circle inside of which perfectly sharp refraction has occurred (Figure 10.12). There is no sense of the sphere as an object as one would experience if scattering due to imperfections had occurred.

As far as light interaction is concerned, distributed ray tracing again only considers specular interaction but this time imperfect specular interaction is simulated by using the ray tracing approach and constructing at every hit point a reflection lobe. The shape of the lobe can depend on the surface properties of the material. Instead of spawning a single transmitted or reflected ray at an intersection a group of rays is spawned which samples the reflection lobe. This produces more realistic ray traced scenes. The images of objects reflected in the surfaces of nearby objects can appeared blurred, transparency effects are more realistic because scattering imperfections can be simulated. Area light sources can be included in the scene to produce shadows. Consider Figure 10.13: if, as would be the case in practice, the mirror surface of the sphere was not physically perfect, then we would expect to see a blurred reflection of the opaque sphere in the mirror sphere.

Thus the path classification scheme is again LDS*E or LS*E but this time all the paths are calculated (or more precisely an estimation of the effects of all the paths is calculated by judicious sampling). In Figure 10.13 three LDSE paths may be discovered by a single eye ray. The points on the wall hit by these rays are combined into a single ray (and eventually a single pixel).

As well as the above effects, Cook *et al.*'s (1984) method considered a finite aperture camera model which produced images that exhibit depth of field, motion blur

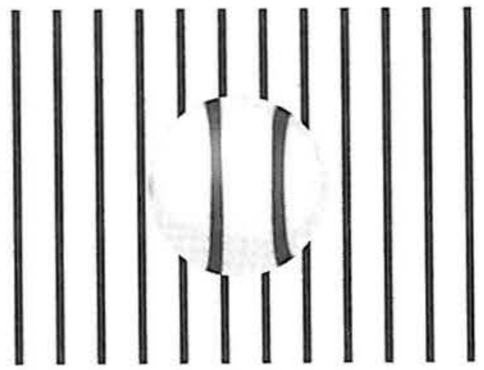


Figure 10.12
Perfect refraction through a solid glass sphere is indistinguishable from texture mapping.

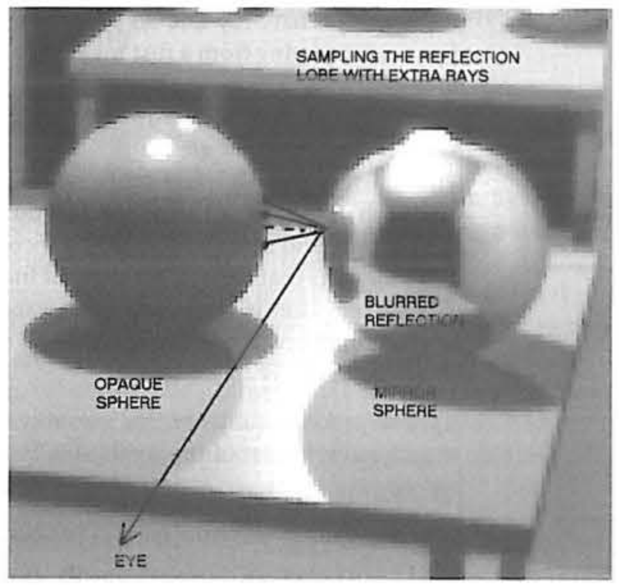


Figure 10.13
Distributed ray tracing for reflection (see Figure 10.4 for the complete geometry of this case).

due to moving objects and effective anti-aliasing (see Chapter 14 for the anti-aliasing implications of this algorithm). Figure 10.14 (Colour Plate) is an image rendered with a distributed ray tracer that demonstrates the depth of field phenomenon. The theoretical importance of this work is their realization that all these phenomena could be incorporated into a single multi-dimensional integral which was then evaluated using Monte Carlo techniques. A ray path in this algorithm is similar to a path in Kajiya's method with the addition of the camera lens. The algorithm uses a combination of stratified and importance sampling. A pixel is stratified into 16 sub-pixels and a ray is initiated from a point within a sub-pixel by using uncorrelated jittering. The lens is also stratified and one stratum on the pixel is associated with a single stratum on the lens (Figure 10.15). Reflection and transmission lobes are importance sampled and the sample point similarly jittered. Cook *et al.* (1984) pre-calculate these and store them in look-up tables associated with a surface type. Each ray derives an index as a function of its position in the

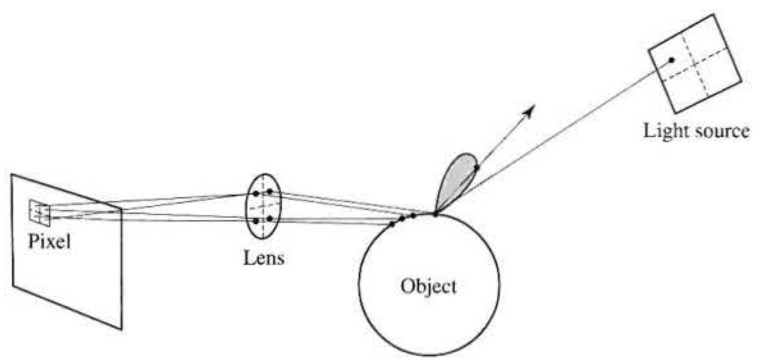


Figure 10.15
Distributed ray tracing: four rays per pixel. The pixel, lens and light source are stratified; the reflection lobe is importance sampled.

pixel. The primary ray and all its descendants have the same index. This means that a ray emerging from a first hit along a direction relative to ***R***, will emerge from all other hits in the same relative ***R*** direction for each object (Figure 10.16). This ensures that each pixel intensity, which is finally determined from 16 samples, is based on samples that are distributed, according to the importance sampling criterion, across the complete range of the specular reflection functions associated with each object. Note that there is nothing to prevent a look-up table being two-dimensional and indexed also by the incoming angle. This enables specular reflection functions that depend on angle of incidence to be implemented. Finally, note that transmission is implemented in exactly the same way using specular transmission functions about the refraction direction.

In summary we have:

(1) The process of distributing rays means that stochastic anti-aliasing becomes an integral part of the method (Chapter 14).
(2) Distributing reflected rays produces blurry reflections.
(3) Distributing transmitted rays produces convincing translucency.
(4) Distributing shadow rays results in penumbrae.
(5) Distributing ray origins over the camera lens area produces depth of field.
(6) Distributing rays in time produces motion blur (temporal anti-aliasing).

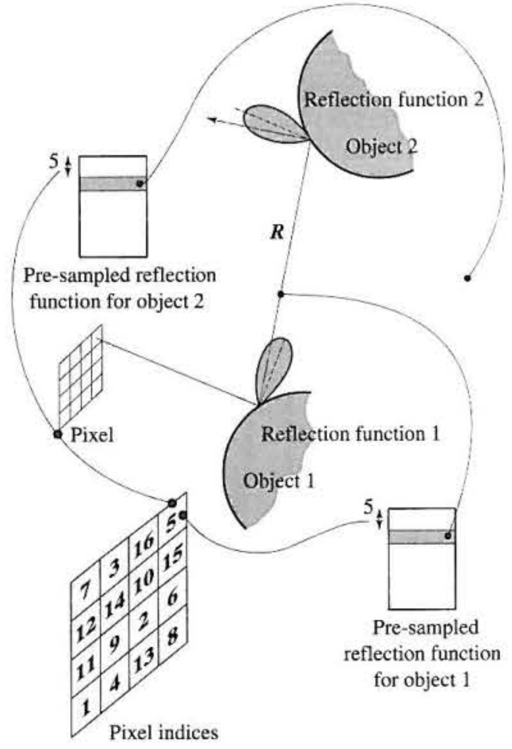


Figure 10.16
Distributed ray tracing and reflected rays.

10.7 Two-pass ray tracing

Two-pass ray tracing (or bi-directional ray tracing) was originally developed to incorporate the specular-to-diffuse transfer mechanism into the general ray tracing model. This accounts for caustics which is the pattern formed on a diffuse surface by light rays being reflected through a medium like glass or water. One can usually see on the bottom and sides of a swimming pool beautiful elliptical patterns of bright light which are due to sunlight refracting at the wind-disturbed water surface causing the light energy to vary across the diffuse surface of the pool sides. Figure 10.17 shows a ray from the scene in Figure 10.4 emanating from the light source refracting through the sphere and contributing to a caustic that forms on the (diffuse) table top. This is an LSSDE path.

Two-pass ray tracing was first proposed by Arvo (1986). In Arvo's scheme, rays from the light source were traced through transparent objects and from specular objects. Central to the working of such a strategy is the question of how information derived during the first pass is communicated to the second. Arvo suggests achieving this with a light or illumination map, consisting of a grid of data points, which is pasted onto each object in the scene in much the same way that a conventional texture map would be.

In general, two-pass ray tracing simulates paths of type LS*DS*E. The algorithm 'relies' on there being a single D interaction encountered from both the light source and the eye. The first pass consists of shooting rays from the light source and following them through the specular interactions until they hit a diffuse surface (Figure 10.17). The light energy from each ray is then deposited or cached on the diffuse surface, which has been subdivided in some manner, into elements or

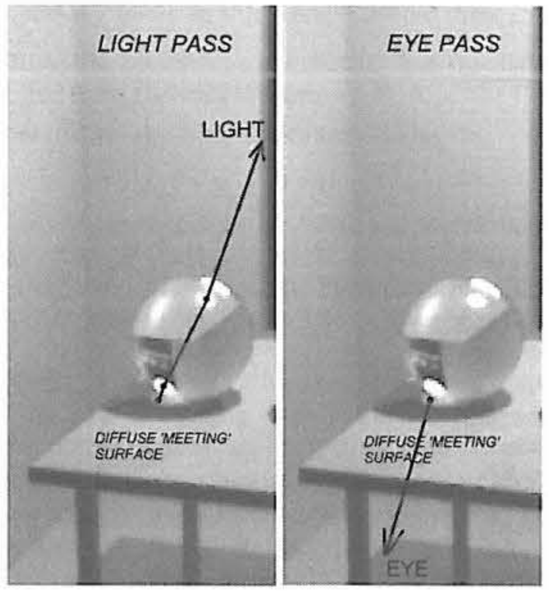


Figure 10.17
Two-pass ray tracing for the LSSDE path in Figure 10.4.

bins. In effect the first pass imposes a texture map or illumination map – the varying brightness of the caustic – on the diffuse surface. The resolution of the illumination map is critical. For a fixed number of shot light rays, too fine a map may result in map elements receiving no rays and too coarse a map results in blurring.

The second pass is the eye trace – conventional Whitted ray tracing – which terminates on the diffuse surface and uses the stored energy in the illumination map as an approximation to the light energy that would be obtained if diffuse reflection was followed in every possible direction from the hit point. In the example shown, the second pass simulates a DE path (or ED path with respect to the trace direction). The 'spreading' of the illumination from rays traced in the first pass over the diffuse surface relies on the fact that the rate of change of diffuse illumination over a surface is slow. It is important to note that there can only be one diffuse surface included in any path. Both the eye trace and the light trace terminate on the diffuse surface – it is the 'meeting point' of both traces.

It is easy to see that we cannot simulate LS*D paths by eye tracing alone. Eye rays do not necessarily hit the light and we have no way of finding out if a surface has received extra illumination due to specular to diffuse transfer. This is illustrated for an easy case of an LSDE path in Figure 10.18.

The detailed process is illustrated in Figure 10.19. A light ray strikes a surface at ***P*** after being refracted. It is indexed into the light map associated with the

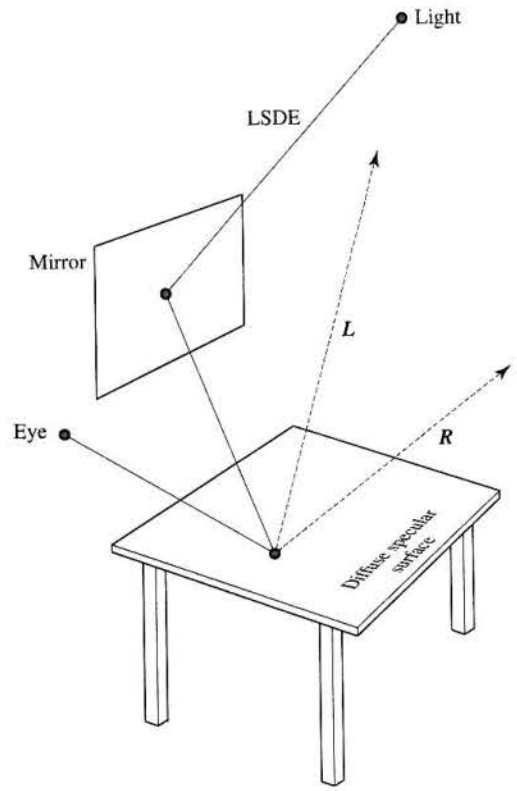


Figure 10.18
An example of an LSDE path (see also Figures 10.4 and 10.17 for examples of SDE paths). An eye ray can 'discover' light ray *L* and reflected ray *R* but cannot find the LSDE path.

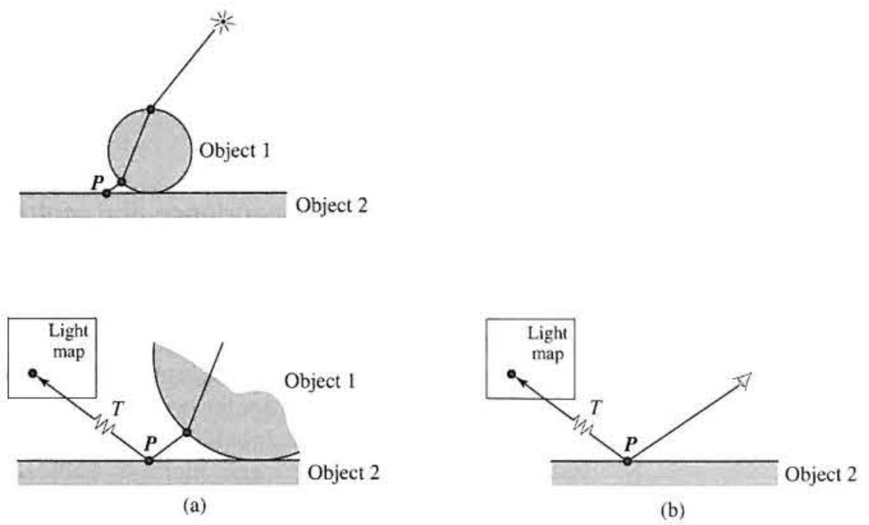


Figure 10.19
Two-pass ray tracing and light maps. (a) First pass: light is deposited in a light map using a standard texture mapping *T*.
(b) Second pass: when object 2 is conventionally eye traced extra illumination at ***P*** is obtained by indexing the light map with *T*.

object using a standard texture mapping function ***T***. During the second pass an eye ray hits ***P***. The same mapping function is used to pick up any illumination for the point ***P*** and this contribution weights the local intensity calculated for that point.

An important point here is that the first pass is view independent – we construct a light map for each object which is analogous in this sense to a texture map – it becomes part of the surface properties of the object. We can use the light maps from any view point after they are completed and they need only be computed once for each scene.

Figure 10.20(a) and (b) (Colour Plate) shows the same scene rendered using a Whitted and two-pass ray tracer. In this scene there are three LSD paths:

(1) Two caustics from the red sphere – one directly from the light and one from the light reflected from the curved mirror.

(2) One (cusp) reflected caustic from the cylindrical mirror.

(3) Secondary illumination from the planar mirror (a non-caustic LSDE path).

Figures 10.20(c)–(e) were produced by shooting an increasing number of light rays and show the effect of the light sprinkled on the diffuse surface. As the number of rays in the light pass increases, these can eventually be merged to form well-defined LSD paths in the image. The number of rays shot in the light pass was 200, 400 and 800 respectively.

Two-pass ray tracing, as introduced by Arvo (1986) was apparently the first algorithm to use the idea of caching illumination. This approach has subsequently been taken up by Ward in his RADIANCE renderer and is the basis of most recent approaches to global illumination (see Section 10.9). We have also introduced the idea of light maps in Chapter 8. The difference between light maps in the context of this chapter and those in Chapter 8 is in their application. In global illumination they are used as part of the rendering process to make a solution more efficient or feasible. Their application in Chapter 8 was as a mechanism for by-passing light calculations in real time rendering. In that context they function as a means of carrying pre-calculated rendering operations into the real time application.

10.8 View dependence/independence and multi-pass methods

In Section 10.5 we introduced path tracing as a method that implemented full L(D|S)* E interaction but pointed out that this is an extremely costly approach to solving the global illumination problem. In this section we will look at approaches which have combined established partial solutions such as ray tracing and radiosity and these are termed multi-pass methods.

A multi-pass method in global illumination most commonly means a combination of a view-independent method (radiosity) with a view-dependent method (ray tracing). (Although we could categorize two-pass ray tracing as a multi-pass method we have chosen to consider it as an extension to ray tracing.) Consider first the implications of the difference between a view-dependent and a view-independent approach. View-independent solutions normally only represent view-independent interactions (pure diffuse–diffuse) because they are mostly solutions where the light levels at every point in the scene are written into a three-dimensional scene data structure. We should bear in mind, however, that in principle there is nothing to stop us computing a view-independent solution that stores specular interactions, we would simply have to increase the dimensionality of the solution to calculate/store the direction of the light on a surface as well as its intensity. We return to this point in Section 10.11.

A pure view-independent algorithm evaluates only sufficient global illumination to determine the final image and if a different view is required the algorithm starts all over again. This is obvious. A more subtle point is that, in general, view-dependent algorithms evaluate an independent solution for each pixel. This is wasteful in the case of diffuse interaction because the illumination on large diffuse surfaces changes only slowly. It was this observation that led to the idea of caching illumination.

View-independent algorithms on the other hand are generally more expensive and do not handle high frequency changes such as specular interaction without significant cost in terms of computation and storage. Multi-pass algorithms exploit the advantages of both approaches by combining them.

A common approach is to post-process a radiosity solution with a ray-tracing pass. A view-independent image with the specular detail added is then obtained. However, this does not account for all path types. By combining radiosity with two-pass ray tracing the path classification, LS*DS*E can be extended to LS*(D*)S*E, the inclusion of radiosity extending the D component to D*. This implies the following ordering for an extended radiosity algorithm. Light ray tracing is employed first and light rays are traced from the source(s) through all specular transports until a diffuse surface is reached and the light energy is deposited. This accounts for the LS* paths. A radiosity solution is then invoked using these values as emitting patches and the deposited energy is distributed through the D* chain. Finally an eye pass is initiated and this provides the final projection and the ES* or ES*D paths.

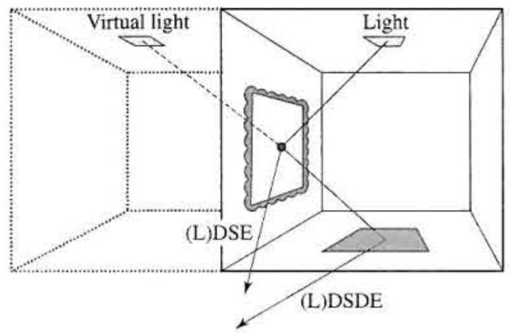


Figure 10.21 The virtual environment method for incorporating DSD paths in the radiosity method.

Comparing the string LS*(D*)S*E with the complete global solution, we see that the central D* paths should be extended to (D*S*D*)* to make LS*(D*S*D*)*S*E which is equivalent to the complete global solution L(S|D)*E. Conventional or classical radiosity does not include diffuse-to-diffuse transfer that takes place via an intermediate specular surface. In other words once we invoke the radiosity phase we need to include the possibility of transfer via an intermediate specular path DSD.

The first and perhaps the simplest approach to including a specular transfer into the radiosity solution was based on modifying the classical radiosity algorithm for flat specular surfaces, such as mirrors, and is called the virtual window approach. This idea is shown in Figure 10.21. Conventional radiosity calculates the geometric relationship between the light source and the floor and the LDE path is accounted for by the diffuse–diffuse interaction between these two surfaces. (Note that since the light source is itself an emitting diffuse patch we can term the path LDE or DDE). What is missing from this is the contribution of light energy from the LSD or DSD path that would deposit a bright area of light on the floor. The DSD path from the light source via the mirror to the floor can be accounted for by constructing a virtual environment 'seen' through the mirror as a window. The virtual light source then acts as if it was the real light source reflected from the mirror. However, we still need to account for the LSE path which is the detailed reflected image formed in the mirror. This is view dependent and is determined during a second pass ray tracing phase. The fact that this algorithm only deals with what is, in effect, a special case illustrates the inherent difficulty of extending radiosity to include other transfer mechanisms.

10.9 Caching illumination

Caching illumination is the term we have given to the scheme of storing three- or five-dimensional values for illumination, in a data structure associated with the scene, as a solution progresses. Such a scheme usually relates to view-dependent algorithms. In other words the cached values are used to speed up or increase the accuracy of a solution; they do not comprise a view-dependent solution in their own right. We can compare such an approach with a view-independent solution such as radiosity where final illumination values are effectively cached on the (discretized) surfaces themselves. The difference between such an approach and the caching methods described in this section is that the storage method is independent of the surface. This means that the meshing problems inherent in surface discretization methods (Chapter 11) are avoided. Illumination values on surfaces are stored in a data structure like an octree which represents the entire three-dimensional extent of the scene.

Consider again the simplified form of the radiance equation:

$$L_{\text{surface}} = \int \rho L_{\text{in}}$$

The BRDF is known but L_{in} is not and this, as we pointed out in Section 10.4, limits the efficacy of importance sampling. An estimate of L_{in} can be obtained as the solution proceeds and this requires that the values are stored. The estimate can be used to improve importance sampling and this is the approach taken by Lafortune and Williams (1995) in a technique that they call adaptive importance sampling. Their method is effectively a path tracing algorithm which uses previously calculated values of radiance to guide the current path. The idea is shown in Figure 10.22 where it is seen that a reflection direction during a path trace is chosen according to both the BRDF for the point and the current value of the field radiance distribution function for that point which has been built up

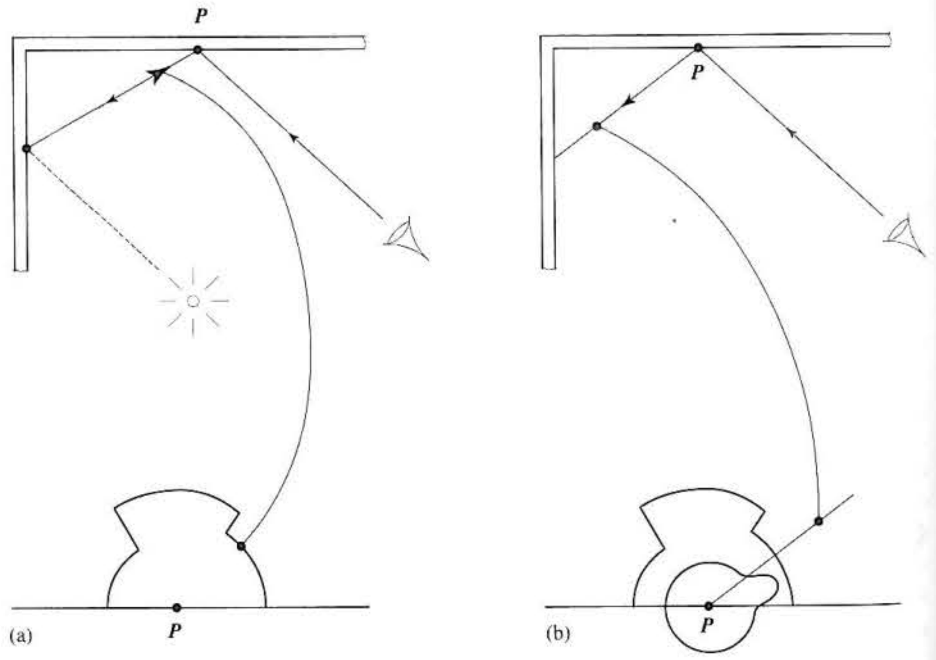


Figure 10.22 Adaptive importance sampling in path tracing (after Lafortune and Williams (1995)).
(a) Incoming radiance at a point ***P*** is cached in a 5D tree and builds up into a distribution function.
(b) A future reflected direction from ***P*** is selected on the basis of both the BRDF and the field radiance distribution function.

from previous values. Lafortune and Williams (1995) cache radiance values in a five-dimensional tree – a two-dimensional extension of a (three-dimensional) octree.

The RADIANCE renderer is probably the most well-known global illumination renderer. Developed by Ward (1994) over a period of nine years, it is a strategy, based on path tracing, that solves a version of the rendering equation under most conditions. The emphasis of this work is firmly on the accuracy required for architectural simulations under a variety of lighting conditions varying from sunlight to complex artificial lighting set-ups. The algorithm is effectively a combination of deterministic and stochastic approaches and Ward (1994) describes the underlying motivations as follows:

The key to fast convergence is in deciding *what* to sample by removing those parts of the integral we can compute deterministically and gauging the importance of the rest so as to maximise the payback from our ray calculations.

Specular calculations are made separately and the core algorithm deals with indirect diffuse interaction. Values resulting from (perfect) diffuse interaction are cached in a (three-dimensional) octree and these cached values are used to interpolate a new value if a current hit point is sufficiently close to a cached point. This basic approach is elaborated by determining the 'irradiance gradient' in the region currently being examined which leads to the use of a higher-order (cubic) interpolation procedure for the interpolation. The RADIANCE renderer is a path tracing algorithm that terminates early if the cached values are 'close enough'.

Finally Ward expresses some strong opinions about the practical efficacy of the radiosity method. It is unusual for such criticisms to appear in a computer graphics paper and Ward is generally concerned that the radiosity method has not migrated from the research laboratories. He says:

For example, most radiosity systems are not well automated, and do not permit general reflectance models or curved surfaces . . .. Acceptance of physically based rendering is bound to improve, but researchers must first demonstrate the real-life applicability of their techniques. There have been few notable successes in applying radiosity to the needs of practising designers. While much research has been done on improving efficiency of the basic radiosity method, problems associated with more realistic complicated geometries, have only recently got the attention they deserve. For whatever reason it appears that radiosity has yet to fulfil its promise, and it is time to re-examine this technique in the light of real-world applications and other alternatives for solving the rendering equation.

An example of the use of the RADIANCE renderer is given in the comparative image study in Chapter 18 (Figure 18.19).

10.10 Light volumes

Light volume is the term given to schemes that cache a view-independent global illumination by storing radiance or irradiance values at sample points over all space (including empty space). Thus they differ from the previous schemes which stored values only at point on surfaces. In Chapter 16 we also encounter light volumes (therein termed light fields). Light fields are the same as light volumes and differ only in their intended application. They are used to efficiently store a pre-calculated view-independent rendering of a scene. In global illumination, however, they are used to facilitate a solution in some way.

An example of the application of light volumes is described by Greger *et al.* (1998). Here the idea is to use a global illumination solution in 'semi-dynamic' environments. Such an environment is defined as one wherein the moving objects are small compared to the static objects, implying that their position in the scene does not affect the global illumination solution to any great extent. A global solution is built up in a light volume and this is used to determine the global illumination received by a moving object – in effect the moving object travels through a static light volume receiving illumination but not contributing to the pre-processed solution.

10.11 Particle tracing and density estimation

In this final section we look at a recent approach (Walter *et al.* 1997) whose novelty is to recognize that it is advantageous to separate the global problem into light transport and light representation calculations. In this work rather than caching illumination, particle histories are stored. The reason for this is that light transport – the flow of light between surfaces – has high global or inter-surface complexity. On the other hand the representation of light on the surface of an object has high local or intra-surface complexity. Surfaces may exhibit shadows, specular highlights caustics etc. (A good example of this is the radiosity algorithm where transport and representation are merged into one process. Here the difficulty lies in predicting the meshing required – for example, to define shadow edges – to input into the algorithm. That is, we have to decide on a meshing prior to the light transport solution being available.)

The process is divided into three sequential phases:

- **Particle tracing** In a sense this is a view-independent form of path tracing. Light-carrying particles are emitted from each light source and travel through the environment. Each time a surface is hit this is recorded (surface identifier, point hit and wavelength of the particle) and a reflection/transmission direction computed according to the BRDF of the surface. Each particle generates a list of information for every surface it collides with and a particle process terminates after a minimum of interactions or until it is absorbed. Thus a particle path generates a history of interactions rather than returning a pixel intensity.
- **Density estimation** After the particle tracing process is complete each surface possesses a list of particles and the stored hit points are used to construct the illumination on the surface based on the spatial density of the hit points. The result of this process is a Gouraud shaded mesh of triangles.

- **Mesh optimization** The solution is view independent and the third phase optimizes or decimates the mesh by progressively removing meshes as long as the resulting change due to the removal does not drop below a (perceptually based) threshold. The output from this phase is an irregular mesh whose detail relates to the variation of light over the surface.

Walter *et al.* (1997) point out that a strong advantage of the technique is that its modularity enables optimization for different design goals. For example, the light transport phase can be optimized for the required accuracy of the BRDFs. The density phase can vary its criteria according to perceptual accuracy and the decimation phase can achieve high compression while maintaining perceptual quality.

A current disadvantage of the approach is that it is a three-dimensional view-independent solution which implies that it can only display diffuse–diffuse interaction. However, Walter *et al.* (1997) point out that this restriction comes out of the density estimation phase. The particle tracing module can deal with any type of BRDF.

11 The radiosity method

Introduction

Ray tracing, the first computer graphics model to embrace global interaction, or at least one aspect of it – suffers from an identifying visual signature: you can usually tell if an image has been synthesized using ray tracing. It only models one aspect of the light interaction – that due to perfect specular reflection and transmission. The interaction between diffusely reflecting surfaces, which tends to be the predominant light transport mechanism in interiors, is still modelled using an ambient constant (in the local reflection component of the model). Consider, for example, a room with walls and ceiling painted with a matte material and carpeted. If there are no specularly reflecting objects in the room, then those parts of the room that cannot see a light source are lit by diffuse interaction. Such a room tends to exhibit slow and subtle changes of intensity across its surfaces.

In 1984, using a method whose theory was based on the principles of radiative heat transfer, researchers at Cornell University, developed the radiosity method (Goral *et al.* 1984). This is now known as classical radiosity and it simulates LD*E paths, that is, it can only be used, in its unextended form, to render scenes that are made up in their entirety of (perfect) diffuse surfaces.

To accomplish this, every surface in a scene is divided up into elements called patches and a set of equations is set up based on the conservation of light energy.

A single patch in such an environment reflects light received from every other patch in the environment. It may also emit light if it is a light source – light sources are treated like any other patch except that they have non-zero self-emission. The interaction between patches depends on their geometric relationship. That is distance and relative orientation. Two parallel patches a short distance apart will have a high interaction. An equilibrium solution is possible if, for each patch in the environment, we calculate its interaction between it and every other patch in the environment.

One of the major contributions of the Cornell group was to invent an efficient way – the hemicube algorithm – for evaluating the geometric relationship between pairs of patches; in fact, in the 1980s most of the innovations in radiosity methods have come out of this group.

The cost of the algorithm is $O(N^2)$ where N is the number of patches into which the environment is divided. To keep processing costs down, the patches are made large and the light intensity is assumed to be constant across a patch. This immediately introduces a quality problem – if illumination discontinuities do not coincide with patch edges artefacts occur. This size restriction is the practical reason why the algorithm can only calculate diffuse interaction, which by its nature changes slowly across a surface. Adding specular interaction to the radiosity method is expensive and is still the subject of much research. Thus we have the strange situation that the two global interaction methods – ray tracing and radiosity – are mutually exclusive as far as the phenomena that they calculate are concerned. Ray tracing cannot calculate diffuse interaction and radiosity cannot incorporate specular interaction. Despite this, the radiosity method has produced some of the most realistic images to date in computer graphics.

The radiosity method deals with shadows without further enhancement. As we have already discussed, the geometry of shadows is more-or-less straightforward to calculate and can be part of a ray tracing algorithm or an algorithm added onto a local reflection model renderer. However, the intensity within a shadow is properly part of diffuse interaction and can only be arbitrarily approximated by other algorithms. The radiosity method takes shadows in its stride. They drop out of the solution as intensities like any other. The only problem is that the patch size may have to be reduced to delineate the shadow boundary to some desired level of accuracy. Shadow boundaries are areas where the rate of change of diffuse light intensity is high and the normal patch size may cause visible aliasing at the shadow edge.

The radiosity method is an object space algorithm, solving for the intensity at discrete points or surface patches within an environment and not for pixels in an image plane projection. The solution is thus independent of viewer position. This complete solution is then injected into a renderer that computes a particular view by removing hidden surfaces and forming a projection. This phase of the method does not require much computation (intensities are already calculated) and different views are easily obtained from the general solution.

11.1 Radiosity theory

Elsewhere in the text we have tried to maintain a separation between the algorithm that implements a method and the underlying mathematics. It is the case, however, that with the radiosity method, the algorithm is so intertwined with the mathematics that it would be difficult to try to deal with this in a separate way. The theory itself consists of nothing more than definitions – there is no manipulation. Readers requiring further theoretical insight are referred to the book by Siegel and Howell (1984).

The radiosity method is a conservation of energy or energy equilibrium approach, providing a solution for the radiosity of all surfaces within an enclosure. The energy input to the system is from those surfaces that act as emitters. In fact, a light source is treated like any other surface in the algorithm except that it possesses an initial (non-zero) radiosity. The method is based on the assumption that all surfaces are perfect diffusers or ideal Lambertian surfaces.

Radiosity, B, is defined as the energy per unit area leaving a surface patch per unit time and is the sum of the emitted and the reflected energy:

$$B_i\mathrm{d}A_i = E_i\mathrm{d}A_i + R_i\int_j B_jF_{ji}\mathrm{d}A_j$$

Expressing this equation in words we have for a single patch i:

radiosity × area = emitted energy + reflected energy

E_i is the energy emitted from a patch. The reflected energy is given by multiplying the incident energy by R_i, the reflectivity of the patch. The incident energy is that energy that arrives at patch i from all other patches in the environment; that is we integrate over the environment, for all j ($j \neq i$), the term $B_jF_{ji}\mathrm{d}A_j$. This is the energy leaving each patch j that arrives at patch i. F_{ji} is a constant, called a form factor, that parametrizes the relationship between patches j and i.

We can use a reciprocity relationship to give:

$$F_{ij}A_i = F_{ji}A_j$$

and dividing through by $\mathrm{d}A_i$, gives:

$$B_i = E_i + R_i\int_j B_jF_{ij}$$

For a discrete environment the integral is replaced by a summation and constant radiosity is assumed over small discrete patches, giving:

$$B_i = E_i + R_i\sum_{j=1}^{n} B_jF_{ij}$$

Such an equation exists for each surface patch in the enclosure and the complete environment produces a set of n simultaneous equations of the form:

$$\begin{bmatrix} 1-R_1F_{11} & -R_1F_{12} & \dots & -R_1F_{1n} \\ -R_2F_{21} & 1-R_2F_{22} & \dots & -R_2F_{2n} \\ \vdots & \vdots & \dots & \vdots \\ R_nF_{n1} & -R_nF_{n2} & \dots & 1-R_nF_{nn} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{bmatrix} \qquad [11.1]$$

Solving this equation is the radiosity method. Out of this solution comes B_i the radiosity for each patch. However, there are two problems left. We need a way of computing the form factors. And we need to compute a view and display the patches. To do this we need a linear interpolation method – just like Gouraud shading – otherwise the subdivision pattern – the patches themselves – will be visible.

The E_is are non-zero only at those surfaces that provide illumination and these terms represent the input illumination to the system. The R_is are known and the F_{ij}s are a function of the geometry of the environment. The reflectivities are wavelength-dependent terms and the above equation should be regarded as a monochromatic solution; a complete solution being obtained by solving for however many colour bands are being considered. We can note at this stage that $F_{ii} = 0$ for a plane or convex surface – none of the radiation leaving the surface will strike itself. Also from the definition of the form factor the sum of any row of form factors is unity.

Since the form factors are a function only of the geometry of the system they are computed once only. The method is bound by the time taken to calculate the form factors expressing the radiative exchange between two surface patches A_i and A_j. This depends on their relative orientation and the distance between them and is given by:

$$F_{ij} = \frac{\text{Radiative energy leaving surface } A_i \text{ that strikes } A_j \text{ directly}}{\text{Radiative energy leaving surface } A_i \text{ in all directions in the hemispherical space surrounding } A_i}$$

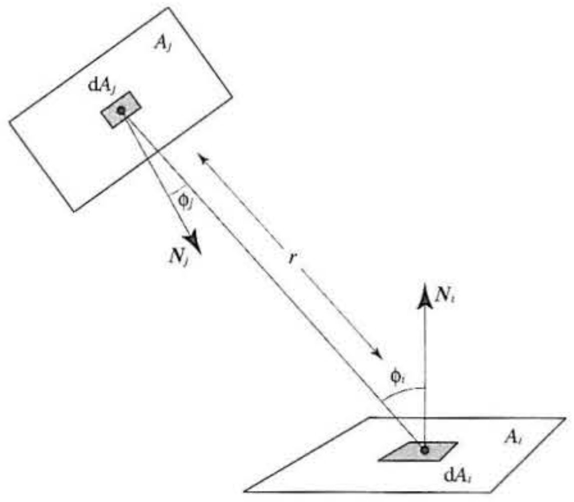


Figure 11.1 Form factor geometry for two patches i and j (after Goral *et al.* (1984)).

It can be shown that this is given by:

$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos \phi_i \cos \phi_j}{\pi r^2} \, dA_j dA_i$$

where the geometric conventions are illustrated in Figure 11.1. In any practical environment A_j may be wholly or partially invisible from A_i and the integral needs to be multiplied by an occluding factor which is a binary function that depends on whether the differential area dA_i can see dA_j or not. This double integral is difficult to solve except for specific shapes.

11.2 Form factor determination

An elegant numerical method of evaluating form factors was developed in 1985 and this is known as the hemicube method. This offered an efficient method of determining form factors and at the same time a solution to the intervening patch problem.

The patch to patch form factor can be approximated by the differential area to finite area equation:

$$F_{dA_iA_j} = \int_{A_j} \frac{\cos \phi_i \cos \phi_j}{\pi r^2} \, dA_j$$

where we are now considering the form factor between the elemental area dA_i and the finite area A_j. dA_i is positioned at the centre point of patch i. The veracity of this approximation depends on the area of the two patches compared with the distance, r, between them. If r is large the inner integral does not change much over the range of the outer integral and the effect of the outer integral is simply multiplication by unity.

A theorem called the Nusselt analogue tells us that we can consider the projection of a patch j onto the surface of a hemisphere surrounding the elemental patch dA_i and that this is equivalent in effect to considering the patch itself. Also patches that produce the same projection on the hemisphere have the same form factor. This is the justification for the hemicube method as illustrated in Figure 11.2. Patches A, B and C all have the same form factor and we can evaluate the form factor of any patch j by considering not the patch itself, but its projection onto the faces of a hemicube.

A hemicube is used to approximate the hemisphere because flat projection planes are computationally less expensive. The hemicube is constructed around the centre of each patch with the hemicube Z axis and the patch normal coincident (Figure 11.3). The faces of the hemicube are divided into pixels – a somewhat confusing use of the term since we are operating in object space. Every other patch in the environment is projected onto this hemicube. Two patches that project onto the same pixel can have their depths compared and the further patch be rejected, since it cannot be seen from the receiving patch. This approach is analogous to a Z-buffer algorithm except that there is no interest in intensities at this stage. The hemicube algorithm only facilitates the calculation of the form factors that are subsequently used in calculating diffuse intensities and a 'label buffer' is maintained indicating which patch is currently nearest to the hemicube pixel.

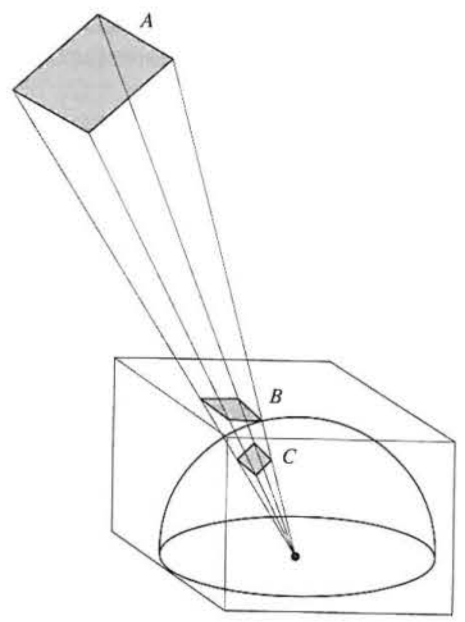


Figure 11.2
The justification for using a hemicube. Patches A, B and C have the same form factor.

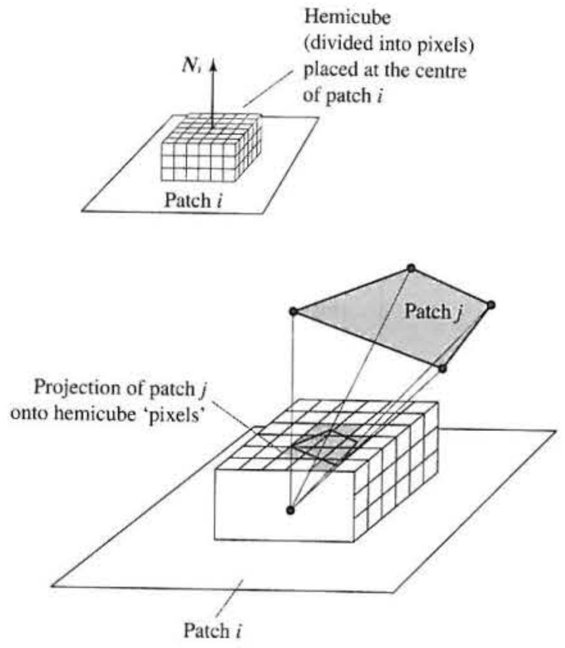

Hemicube (divided into pixels) placed at the centre of patch i
N_i
Patch i
Patch j
Projection of patch j onto hemicube 'pixels'
Patch i

Figure 11.3
Evaluating the form factor F_{ij} by projecting patch j onto the faces of a hemicube centred on patch i.

Each pixel on the hemicube can be considered as a small patch and a differential to finite area form factor, known as a delta form factor, defined for each pixel. The form factor of a pixel is a fraction of the differential to finite area form factor for the patch and can be defined as:

$$\begin{aligned}\Delta F_{dAiAj} &= \frac{\cos \phi_i \cos \phi_j}{\pi r^2} \Delta A \\ &= \Delta F_q\end{aligned}$$

where ΔA is the area of the pixel.

These form factors are pre-calculated and stored in a look-up table. This is the foundation of the efficiency of the hemicube method. Again, using the fact that areas of equal projection onto the receiving surface surrounding the centre of patch A_i have equal form factors, we can conclude that F_{ij}, for any patch, is obtained by summing the pixel form factors onto which patch A_j projects (Figure 11.4).

Thus form factor evaluation now reduces to projection onto mutually orthogonal planes and a summation operation.

Figure 11.5 (Colour Plate) is an interesting image that shows the state of a hemicube placed on the window (Figure 10.7) after all other patches in the scene have been projected onto it. A colour identifies each patch in the scene (and every partial patch) that can be seen by this hemicube. The algorithm then simply summates all the hemicube element form factors associated with each patch.

The method can be summarized in the following stages:

(1) Computation of the form factors, F_{ij}. Each hemicube emplacement calculates $(n-1)$ form factors or one row in the equation.

(2) Solving the radiosity matrix equation.

(3) Rendering by injecting the results of stage (2) into a bilinear interpolation scheme.

(4) Repeating stages (2) and (3) for the colour bands of interest.

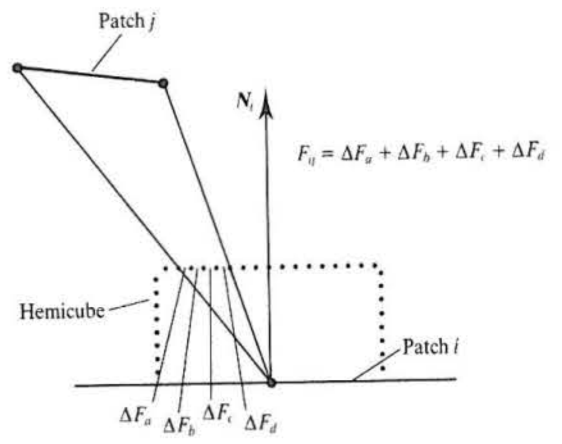


Figure 11.4
F_{ij} is obtained by summing the form factors of the pixels onto which patch i projects.

This process is shown in Figure 11.6. Form factors are a function only of the environment and are calculated once only and can be reused in stage (2) for different reflectivities and light source values. Thus a solution can be obtained for the same environment with, for example, some light sources turned off. The solution produced by stage (2) is a view-independent solution and if a different view is required then only stage (3) is repeated. This approach can be used, for example, when generating an animated walk-through of a building interior. Each frame in the animation is computed by changing the view point and calculating a new view from an unchanging radiosity solution. It is only if we change the geometry of the scene that a re-calculation of the form factors is necessary. If the lighting is changed and the geometry is unaltered, then only the equation needs resolving – we do not have to re-calculate the form factors.

Stage (2) implies the computation of a view-independent rendered version of the solution to the radiosity equation which supplies a single value, a radiosity, for each patch in the environment. From these values vertex radiosities are calculated and these vertex radiosities are used in the bilinear interpolation scheme to provide a final image. A depth buffer algorithm is used at this stage to evaluate the visibility of each patch at each pixel on the screen. (This stage should not be confused with the hemicube operation that has to evaluate inter-patch visibility during the computation of form factors.)

The time taken to complete the form factor calculation depends on the square of the number of patches. A hemicube calculation is performed for every patch (onto which all other patches are projected). The overall calculation time thus depends on the complexity of the environment and the accuracy of the solution,

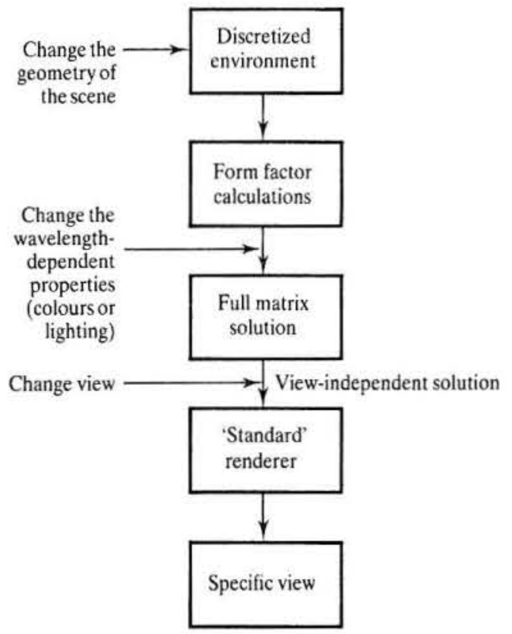

Change the geometry of the scene
Discretized environment
Form factor calculations
Change the wavelength-dependent properties (colours or lighting)
Full matrix solution
Change view
View-independent solution
'Standard' renderer
Specific view

Figure 11.6
Stages in a complete radiosity solution. Also shown are the points in the process where various modifications can be made to the image.

as determined by the hemicube resolution. Although diffuse illumination changes only slowly across a surface, aliasing can be caused by too low a hemicube resolution and accuracy is required at shadow boundaries (see Section 11.7). Storage requirements are also a function of the number of patches required. All these factors mean that there is an upward limit on the complexity of the scenes that can be handled by the radiosity method:

11.3 The Gauss–Siedel method

Cohen and Greenberg (1985) point out that the Gauss–Siedel method is guaranteed to converge rapidly for equation sets such as Equation 11.1. The sum of any row of form factors is by definition less than unity and each form factor is multiplied by a reflectivity of less than one. The summation of the row terms in Equation 11.1 (excluding the main diagonal term) is thus less than unity. The mean diagonal term is always unity ($F_{ii} = 0$ for all i) and these conditions guarantee fast convergence. The Gauss–Siedel method is an extension to the following iterative method. Given a system of linear equations:

$$\boldsymbol{Ax} = \boldsymbol{E}$$

such as Equation 11.1, we can rewrite equations for $x_1, x_2, \ldots, x_i$ in the form:

$$x_1 = \frac{E_1 - a_{12}x_2 - a_{13}x_3 - \ldots - a_{1n}\,x_n}{a_{11}}$$

which leads to the iteration:

$$x_1^{(k+1)} = \frac{E_1 - a_{12}x_2^{(k)} - a_{13}x_3 - \ldots - a_{1n}\,x_n^{(k)}}{a_{11}}$$

in general:

$$x_i^{(k+1)} = \frac{E_i - a_{i1}x_1^{(k)} - \ldots - a_{i,i-1}x_{i-1}^{(k)} - a_{i,i+1}x_{i+1}^{(k)} - \ldots - a_{in}\,x_n^{(k)}}{a_{ii}} \qquad [11.2]$$

This formula can be used in an iteration procedure:

(1) Choose an initial approximation, say:

$$x_i^{(0)} = \frac{E_i}{a_{ii}}$$

for $i = 1, 2, \ldots, n$, where E_i is non-zero for emitting surfaces or light sources only.

(2) Determine the next iterate:

$x_i^{(k+1)}$ from $x_i^{(k)}$

using Equation 11.2.

(3) If $|x_i^{(k+1)} - x_i^{(k)}| <$ a threshold

for $i = 1, 2, \ldots, n$

then stop the iteration, otherwise return to step (2).

This is known as Jacobi iteration. The Gauss–Siedel method improves on the convergence of this method by modifying Equation 11.2 to use the latest available information. When the new iterate $x_i^{(k+1)}$ is being calculated, new values:

$$x_1^{(k+1)}, x_2^{(k+1)}, \ldots, x_{i-1}^{(k+1)}$$

have already been calculated and Equation 11.2 is modified to:

$$x_i^{(k+1)} = \frac{E_i - a_{i1}x_1^{(k+1)} - \ldots - a_{i,i-1}x_{i-1}^{(k+1)} - a_{i,i+1}x_{i+1}^{(k)} - \ldots - a_{in}\,x_n^{(k)}}{a_{ii}} \qquad [11.3]$$

Note that when $i = 1$ the right-hand side of the equation contains terms with superscript k only, and Equation 11.3 reduces to Equation 11.2. When $i = n$ the right-hand side contains terms with superscript $(k+1)$ only.

Convergence of the Gauss–Siedel method can be improved by the following method. Having produced a new value $x_i^{(k+1)}$, a better value is given by a weighted average of the old and new values:

$$x_i^{(k+1)} = rx_i'^{(k+1)} + (1 - r)x_i^{(k)}$$

where r (>0) is a parameter independent of k and i. Cohen *et al.* (1988) report that a relaxation factor of 1.1 works for most environments.

11.4 Seeing a partial solution – progressive refinement

Using the radiosity method in a practical context, such as in the design of building interiors, means that the designer has to wait a long time to see a completed image. This is disadvantageous since one of the *raisons d'être* of computer-based design is to allow the user free and fast experimentation with the design parameters. A long feedback time discourages experimentation and stultifies the design process.

In 1988 the Cornell team developed an approach, called 'progressive refinement' that enabled a designer to see an early (but approximate) solution. At this stage major errors can be seen and corrected, and another solution executed. As the solution becomes more and more accurate, the designer may see more subtle changes that have to be made. We introduced this method in the previous chapter, we will now look at the details.

The general goal of progressive or adaptive refinement can be taken up by any slow image synthesis technique and it attempts to find a compromise between the competing demands of interactivity and image quality. A synthesis method that provides adaptive refinement would present an initial quickly rendered image to the user. This image is then progressively refined in a 'graceful' way. This is defined as a progression towards higher quality, greater realism etc., in a way that is automatic, continuous and not distracting to the user. Early availability of an approximation can greatly assist in the development of techniques and images, and reducing the feedback loop by approximation is a necessary adjunct to the radiosity method.

The two major cost factors in the radiosity method are the storage costs and the calculation of the form factors. For an environment of 50×10^3 patches, even although the resulting square matrix of form factors may be 90% sparse (many patches cannot see each other) this still requires 10^9 bytes of storage (at four bytes per form factor).

Both the requirements of progressive refinement and the elimination of pre-calculation and storage of the form factors are met by an ingenious restructuring of the basic radiosity algorithm. The stages in the progressive refinement are obtained by displaying the results as the iterative solution progresses. The solution is restructured and the form factor evaluation order is optimized so that the convergence is 'visually graceful'. This restructuring enables the radiosity of all patches to be updated at each step in the solution, rather than a step providing the solution for a single patch. Maximum visual difference between steps in the solution can be achieved by processing patches according to their energy contribution to the environment. The radiosity method is particularly suited to a progressive refinement approach because it computes a view-independent solution. Viewing this solution (by rendering from a particular view point) can proceed independently as the radiosity solution progresses.

In the conventional evaluation of the radiosity matrix (using, for example, the Gauss–Seidel method) a solution for one row provides the radiosity for a single patch i:

$$B_i = E_i + R_i \sum_{j=1}^{n} B_j F_{ij}$$

This is an estimate of the radiosity of patch i based on the current estimate of all other patches. This is called 'gathering'. The equation means that (algorithmically) for patch i we visit every other patch in the scene and transfer the appropriate amount of light from each patch j to patch i according to the form factor. The algorithm proceeds on a row-by-row basis and the entire solution is updated for one step through the matrix (although the Gauss–Seidel method uses the new values as soon as they are computed). If the process is viewed dynamically, as the solution proceeds, each patch intensity is updated according to its row position in the radiosity matrix. Light is gathered from every other patch in the scene and used to update the single patch currently being considered.

The idea of the progressive refinement method is that the entire image of all patches is updated at every iteration. This is termed 'shooting', where the contribution from each patch i is distributed to all other patches. The difference between these two processes is illustrated diagramatically in Figures 11.7(a) and (b). This re-ordering of the algorithm is accomplished in the following way.

A single term determines the contribution to the radiosity of patch j due to that from patch i:

$$B_j \quad \text{due to} \quad B_i = R_j B_i F_{ji}$$

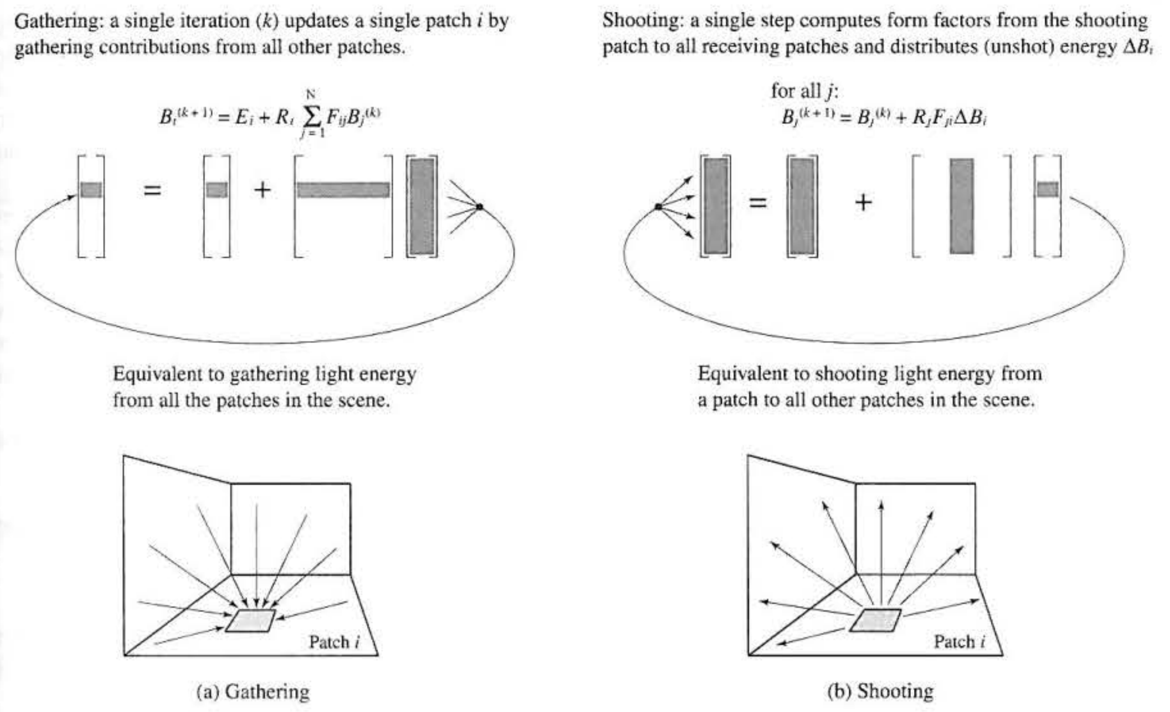

Gathering: a single iteration (k) updates a single patch i by gathering contributions from all other patches.
$B_i^{(k+1)} = E_i + R_i \sum_{j=1}^{N} F_{ij} B_j^{(k)}$
=
+
Equivalent to gathering light energy from all the patches in the scene.
Patch i
(a) Gathering
Shooting: a single step computes form factors from the shooting patch to all receiving patches and distributes (unshot) energy ΔB_i
for all j:
$B_j^{(k+1)} = B_j^{(k)} + R_j F_{ji} \Delta B_i$
=
+
Equivalent to shooting light energy from a patch to all other patches in the scene.
Patch i
(b) Shooting

Figure 11.7
(a) Gathering and
(b) shooting in radiosity solution strategies
(based on an illustration in Cohen *et al.* (1988)).

This relationship can be reversed by using the reciprocity relationship:

$$B_j \quad \text{due to} \quad B_i = R_j B_i F_{ij}\, A_i / A_j$$

and this is true for all patches j. This relationship can be used to determine the contribution to each patch j in the environment from the single patch i. A single radiosity (patch i) shoots light into the environment and the radiosities of all patches j are updated simultaneously. The first complete update (of all the radiosities in the environment) is obtained from 'on the fly' form factor computations. Thus an initial approximation to the complete scene can appear when only the first row of form factors has been calculated. This eliminates high start-up or pre-calculation costs.

This process is repeated until convergence is achieved. All radiosities are initially set either to zero or to their emission values. As this process is repeated for each patch i the solution is displayed and at each step the radiosities for each patch j are updated. As the solution progresses the estimate of the radiosity at a patch i becomes more and more accurate. For an iteration the environment already contains the contribution of the previous estimate of B_i and the so-called 'unshot' radiosity – the difference between the current and previous estimates – is all that is injected into the environment.

If the output from the algorithm is displayed without further elaboration, then a scene, initially dark, gradually gets lighter as the incremental radiosities are added to each patch. The 'visual convergence' of this process can be optimized by sorting the order in which the patches are processed according to the amount of energy that they are likely to radiate. This means, for example, that emitting patches, or light sources, should be treated first. This gives an early well lit solution. The next patches to be processed are those that received most light from the light sources and so on. By using this ordering scheme, the solution proceeds in a way that approximates the propagation of light through an environment. Although this produces a better visual sequence than an unsorted process, the solution still progresses from a dark scene to a fully illuminated scene. To overcome this effect an arbitrary ambient light term is added to the intermediate radiosities. This term is used only to enhance the display and is not part of the solution. The value of the ambient term is based on the current estimate of the radiosities of all patches in the environment, and as the solution proceeds and becomes 'better lit' the ambient contribution is decreased.

Four main stages are completed for each iteration in the algorithm. These are:

(1) Find the patch with the greatest (unshot) radiosity or emitted energy.

(2) Evaluate a column of form factors, that is, the form factors from this patch to every other patch in the environment.

(3) Update the radiosity of each of the receiving patches.

(4) Reduce the temporary ambient term as a function of the sum of the differences between the current values calculated in step (3) and the previous values.

An example of the progressive refinement during execution is shown in Figure 10.7 and Section 10.3.2 contains a full description of this figure.

11.5 Problems with the radiosity method

There are three significant problems associated with radiosity rendering. They are algorithm artefacts that appear in the image, the inability to deal with specular interaction and the inordinate time taken to render a scene of moderate complexity. Curiously, hardly any research effort has been devoted to the time factor, and this is perhaps the reason that radiosity has not generally migrated into applications programs. This contrasts with the situation in ray tracing research in the 1980s, where quite soon after the first ray traced imagery appeared, a large and energetic research effort was devoted to making the method faster. In the remainder of the chapter we will deal exclusively with image quality, noting in passing that it is usually related to execution time – quality can be improved by defining the scene more accurately which in the mainstream method means allowing more iterations in the program.

Developments in the radiosity method beyond the techniques described in the previous chapter have mostly been motivated by defects or artefacts that arise out of the representation of the scene as a set of largish patches. Although other factors, such as taking into account scattering atmospheres and the incorporation of specular reflection are important, addressing the visual defects due to meshing accounts for most research emphasis and it is with this aspect that we will deal.

11.6 Artefacts in radiosity images

The common artefacts in radiosity images that use the classical approach of the previous chapter are due to:

(1) Approximations in the hemicube method for determining the form factors.

(2) Using bilinear interpolation as a reconstruction of the radiosity function from the constant radiosity solution.

(3) Using a meshing or subdivision of the scene that is independent of the nature of the variations in the radiosity function.

The visibility and thus the importance of these depends, of course, on the nature of the scene; but usually the third category is the most noticeable and the most difficult to deal with. In practice the artefacts cannot be treated independently: there is little point in developing a powerful meshing strategy without also dealing with artefacts that emerge from bilinear interpolation. We will now look at these image defects detailing both the cause and the possible cure.

11.6.1 Hemicube artefacts

The serious problem of the hemicube method is aliasing caused by the regular division of the hemicube into uniform pixels. Errors occur as a function of the size of the hemicube pixels due to the assumption that patches will project exactly onto an integer number of pixels, which in general, of course, they do not. This is similar to aliasing in ray tracing. We attempt to gather information from a three-dimensional environment by looking in a fixed number of directions. In ray tracing these directions are given initially by evenly spaced eye-to-pixel rays. In the radiosity method, by projecting the patches onto hemicubes we are effectively sampling with projection rays from the hemicube origin. Figure 11.8 shows a two-dimensional analogue of the problem where a number of identical polygons project onto either one or two pixels depending on the interference between the projection rays and the polygon grid. The polygons are of equal size and equal orientation with respect to patch *i*. Their form factors should be different – because the number of pixels onto which each polygon projects is different for each polygon. However, as the example shows, neighbouring polygons which should have almost equal form factors will produce values in the ratio 2:1.

The geometry of any practical scene can cause problems with the hemicube method. Its accuracy depends on the distance between the patches involved in the calculation. When distances become small the method falls down. This

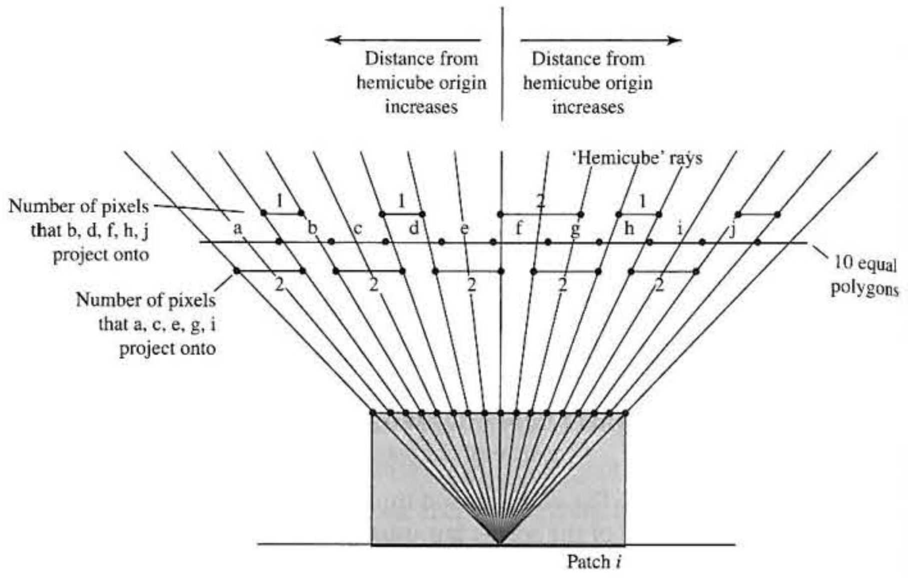

Distance from hemicube origin increases
Distance from hemicube origin increases
'Hemicube' rays
Number of pixels that b, d, f, h, j project onto
10 equal polygons
Number of pixels that a, c, e, g, i project onto
a
b
c
d
e
f
g
h
i
j
1
2
Patch i

Figure 11.8
Interference between hemicube sampling and a set of equal polygons (after Wallace *et al.* (1989)).

situation occurs in practice, for example, when an object is placed on a supporting surface. The errors in form factors occur precisely in those regions from which we expect the radiosity technique to excel and produce subtle phenomena such as colour bleeding and soft shadows. Baum *et al.* (1989) quantify the error involved in form factor determination for proximal surfaces, and demonstrate the hemicube method is only accurate in contexts where the inter-patch distance is at least five patch diameters.

Yet another hemicube problem occurs with light sources. In scenes which the radiosity method is used to render, we are usually concerned with area sources such as fluorescent lights. As with any other surface in the environment we divide the light sources into patches and herein lies the problem. For a standard solution an environment will be discretized into patches where the subdivision resolution depends on the area of surface (and the accuracy of the solution required). However, in the case of light sources the number of hemicubes required or the number of patches required depends on the distance from the closest surface it illuminates. A hemicube operation effectively reduces an emitting patch to a point source. Errors will appear on a close surface as isolated areas of light if the light source is insufficiently subdivided. With strip lights, where the length to breadth ratio is great, insufficient subdivision can give rise to banding or aliasing artefacts that run parallel with the long axis of the light source. An example of the effect of insufficient light source subdivision is shown in Figure 11.14.

Hemicube aliasing can, of course, be ameliorated by increasing the resolution of the hemicube, but this is inefficient, increasing the computation required for all elements in the scene irrespective of whether they are aliased by the hemicube or not; exactly the same situation which occurs with conventional (context independent) anti-aliasing measures (Chapter 14).

Problems emerge from the approximation (see the previous chapter):

$$F_{ij} \approx F_{dA_iA_j}$$

The hemicube evaluates a form factor from a differential area – effectively a point – to a finite area. There are two consequences of this. Figure 11.9 illustrates a problem that can arise with intervening patches. Here the form factor from patch i to patch j is calculated as if the intervening patch did not exist because patch j can be seen in its entirety from the hemicube origin.

Finally, consider the sampling 'efficiency' of the hemicube. Patches that can be 'seen' from the hemicube in the normal direction are more important than patches in the horizon direction. (They project onto hemicube cells that have higher delta form factors.) If we consider distributing the computational effort evenly on the basis of importance sampling then cells nearer the horizon are less important. An investigation reported in Max and Troutman (1993) derives optimal resolution, shapes and grid cell spacings. In this work a top-face resolution 40% higher than that of the sides and a side height of 70% of the width is suggested. Note that this leads also to a reduction in aliasing artefacts caused by uniform hemicube cells.

11.6.2 Reconstruction artefacts

Reconstruction artefacts are so called because they originate from the nature of the method used to reconstruct or approximate the continuous radiosity function from the constant radiosity solution. We recall that radiosity methods can only function under the constant radiosity assumption which is that we divide the environment up into patches and solve a system of equations on the basis that the radiosity is constant across each patch.

The commonest approach – bilinear interpolation – is overviewed in Figure 11.10. Here we assume that the curved surface shown in Figure 11.10(a) will exhibit a continuous variation in radiosity value along the dotted line as shown.

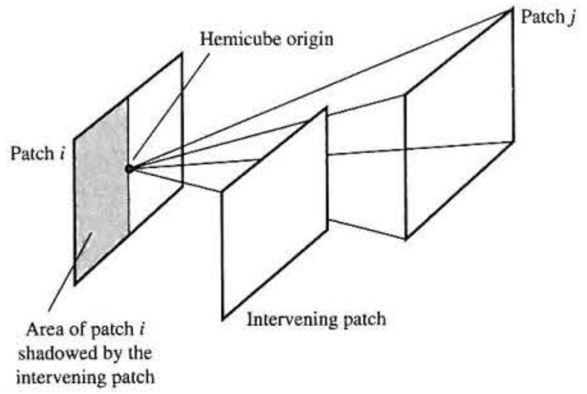


Figure 11.9
All of patch j can be seen from the hemicube origin and the $F_{dA_iA_j}$ approximation falls down.

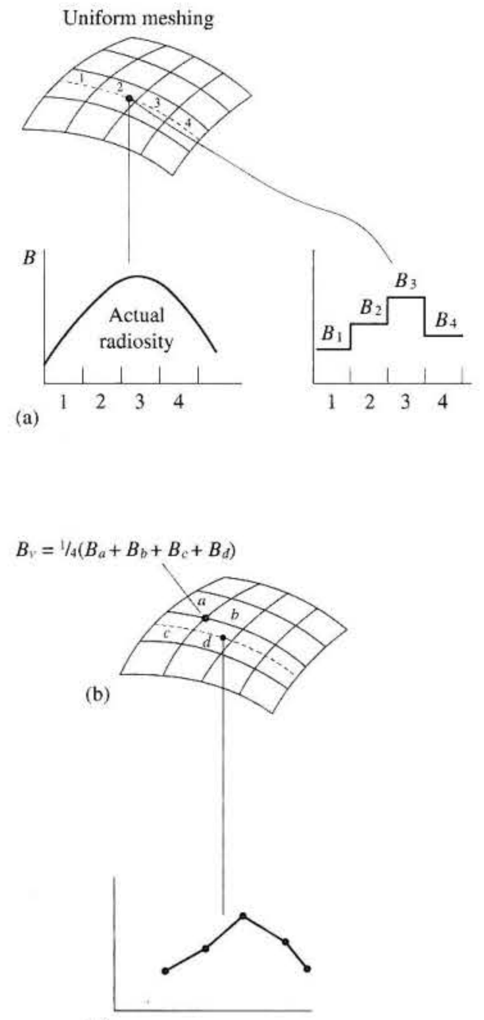


Figure 11.10
Normal reconstruction approach used in the radiosity method.
(a) Compute a constant radiosity solution.
(b) Calculate the vertex radiosities.
(c) Reconstruction by linear interpolation.

The first step in the radiosity method is to compute a constant radiosity solution which will result in a staircase approximation to the continuous function. The radiosity values at a vertex are calculated by averaging the patch radiosities that share the vertex (Figure 11.10(b)). These are then injected into a bilinear interpolation scheme and the surface is effectively Gouraud shaded resulting in the piecewise linear approximation (Figure 11.10(c)).

The most noticeable defect arising out of this process is Mach bands which, of course, we also experience in normal Gouraud shading, where the same interpolation method is used. The 'visual importance' of these can be reduced by using texture mapping but they tend to be a problem in radiosity applications because many of these exhibit large area textureless surfaces – interior walls in buildings, for example. Subdivision meshing strategies also reduce the visibility

of Mach bands because by reducing the size of the elements they reduce the difference between vertex radiosities.

More advanced strategies involve surface interpolation methods (Chapter 3). Here the radiosity values are treated as samples of a continuous radiosity function and quadratic or cubic Bézier/B-spline patch meshes are fitted to these. The obvious difficulties with this approach – its inherent cost and the need to prevent wanted discontinuities being smoothed out – has meant that the most popular reconstruction method is still linear interpolation.

11.6.3 Meshing artefacts

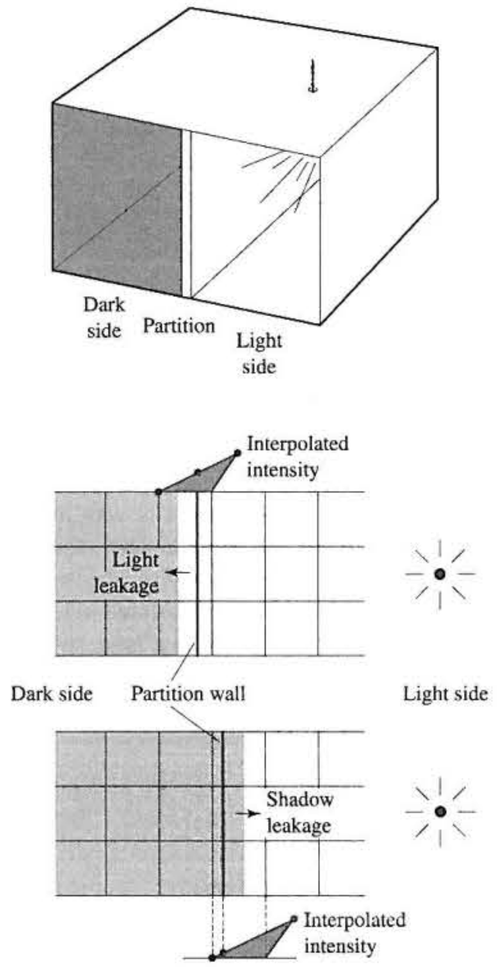
One of the most difficult aspects of the radiosity approach, and one that is still a major research area, is the issue of meshing. In the discussions above we have simply described patches as entities into which the scene is divided with the proviso that these should be large to enable a solution which is not prohibitively expensive. However, the way in which we do this has a strong bearing on the quality of the final image. How should we do this so that the appearance of artefacts is minimized? The reason this is difficult is that we can only do this when we already have a solution, so that we can see where the problems occur. Alternatively we have to predict where the problems will occur and subdivide accordingly. We begin by looking at the nature and origin of meshing artefacts.

First some terminology:

- **Meshing** This is a general term used in the context of radiosity to describe either the initial scene subdivision or the act of further subdivision that may take place while a program is executing. The 'initial scene subdivision' may be a general scene database not necessarily created for input to a radiosity renderer. However, for reasons that will soon become apparent it is more likely to be a preprocessed version of such a database or a scene that has been specifically created for a radiosity solution.
- **Patches** These are the entities in the initial representation of the scene. In a standard radiosity solution, where subdivision occurs during the solution, patches form the input to the program.
- **Elements** These are the portions into which patches are subdivided.

The simplest type of meshing artefact – a so-called D^0 discontinuity – is a discontinuity in the value of the radiosity function. The common sources of such a discontinuity are shadow boundaries caused by a point light source and objects which are in contact. In the former case the light source suddenly becomes visible as we move across a surface and the reconstruction and meshing 'spreads' the shadow edge towards the mesh boundaries. Thus the shadow edge will tend to take the shape of the mesh edges giving it a staircase appearance. However, because we tend to use area light sources in radiosity applications the discontinuities that occur are higher than D^0. Nevertheless these still cause visible

Figure 11.11
Shadow and light leakage.



artefacts. Discontinuities in the derivatives of the radiosity function occur at penumbra and umbra boundaries in shadows in scenes illuminated by area light sources. Again there is 'interference' between the boundaries and the mesh, giving a characteristic 'staircase' appearance to the shadow edges. These are more difficult to deal with than D^0 discontinuities.

When objects are in contact, then unless the intersection boundary coincides with a mesh boundary, shadow or light leakage will occur. The idea is shown in Figure 11.11 for a simple scene. Here, the room is divided by a floor-to-ceiling partition, which does not coincide with patch boundaries on the floor. One half of the room contains a light source and the other is completely dark. Depending on the position of the patch boundaries, the reconstruction will produce either light leakage into the dark region or shadow leakage into the lit region. Figure 18.16 shows the effect of shadow and light leakage for a more complex scene. Despite the fact that the representation contains many more patches than we require for a conventional computer graphics rendering (Gouraud shading) the quality is unacceptably low.

It should be apparent that further subdivision of the scene cannot entirely eliminate shadow and light leakage – it can only reduce it to an acceptable level. It can, however, be eliminated entirely by forcing a meshing along the curve of intersection between objects in contact. Figure 18.18 is the result of meshing the area around a wall light by considering the intersection between the lamp and the wall. Now the wall patch boundaries coincide with the lamp patch boundaries eliminating the leakage that occurred before this meshing.

11.7 Meshing strategies

Meshing strategies that attempt to overcome these defects can be categorized in a number of ways. An important distinction can be made on the basis of when the subdivision takes place:

(1) *A priori* – meshing is completed before the radiosity solution is invoked; that is we predict where discontinuities are going to occur and mesh accordingly. This is also called discontinuity meshing.

(2) *A posteriori* – the solution is initiated with a 'start' mesh which is refined as the solution progresses. This is also called adaptive meshing.

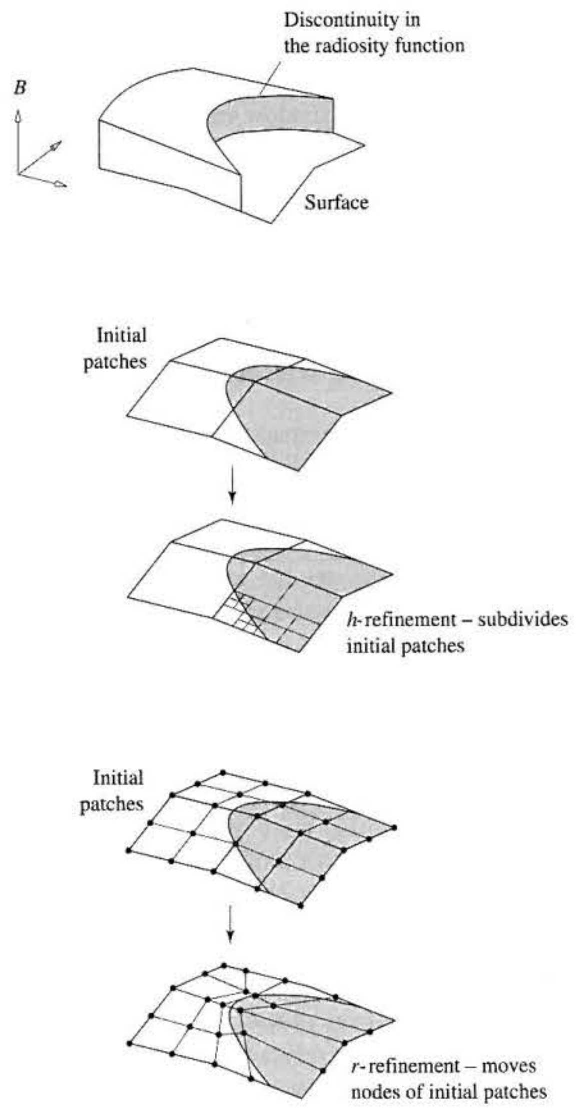
As we have seen, when two objects are in contact, we can eliminate shadow and light leakage by ensuring that mesh element boundaries from each object coincide, which is thus an *a priori* meshing.

Another distinction can be made depending on the geometric nature of the meshing. We can, for example, simply subdivide square patches (non-uniformly) reducing the error to an acceptable level. The commonest approach to date, Cohen and Wallace (1993) term this *h*-refinement. Alternatively we could adopt an approach where the discontinuities in the radiosity function are tracked across a surface and the mesh boundaries placed along the discontinuity boundary. A form of this approach is called *r*-refinement by Cohen and Wallace (1993) where the nodes of the initial mesh are moved in a way that equalizes the error in the elements that share the node. These approaches are illustrated conceptually in Figure 11.12.

11.7.1 Adaptive or *a posteriori* meshing

The classic adaptive algorithm, called substructuring, was described by Cohen *et al.* (1986). Reported before the development of the progressive refinement algorithm, this approach was initially incorporated into a full matrix solution. Adaptive subdivision proceeds by considering the radiosity variation at the nodes or vertices of an element and subdividing if the difference exceeds some threshold.

Figure 11.12
Examples of refinement strategies (*a posteriori*).


Discontinuity in the radiosity function
B
Surface
Initial patches
h-refinement – subdivides initial patches
Initial patches
r-refinement – moves nodes of initial patches

The idea is to generate an accurate solution for the radiosity of a point from the 'global' radiosities obtained from the initial 'coarse' patch computation. Patches are subdivided into elements. Element-to-patch form factors are calculated where the relationship between element-to-patch and patch-to-patch form factors is given by:

$$F_{ij} = \frac{1}{A_i} \sum_{q=1}^{R} F_{(iq)j} A_{(iq)}$$

where:

F_{ij} is the form factor from patch i to patch j
$F_{(iq)j}$ is the form factor from element q of patch i to patch j
$A_{(iq)}$ is the subdivided area of element q of patch i
R is the number of elements in patch i

Patch form factors obtained in this way are then used in a standard radiosity solution.

This increases the number of form factors from $N \times N$ to $M \times N$, where M is the total number of elements created, and naturally increases the time spent in form factor calculation. Patches that need to be divided into elements are revealed by examining the graduation of the coarse patch solution. The previously calculated (coarse) patch solution is retained and the fine element radiosities are then obtained from this solution using:

$$B_{iq} = E_q + R_q \sum_{j=1}^{N} B_j F_{(iq)j} \qquad [11.4]$$

where:

B_{iq} is the radiosity of element q
B_j is the radiosity of patch j
$F_{(iq)j}$ is the element q to patch j form factor

In other words, as far as the radiosity solution is concerned, the cumulative effect of elements of a subdivided patch is identical to that of the undivided patch; or, subdividing a patch into elements does not affect the amount of light that is reflected by the patch. So after determining a solution for patches, the radiosity within a patch is solved independently among patches. In doing this, Equation 11.4 assumes that only the patch in question has been subdivided into elements – all other patches are undivided. The process is applied iteratively until the desired accuracy is obtained. At any step in the iteration we can identify three stages:

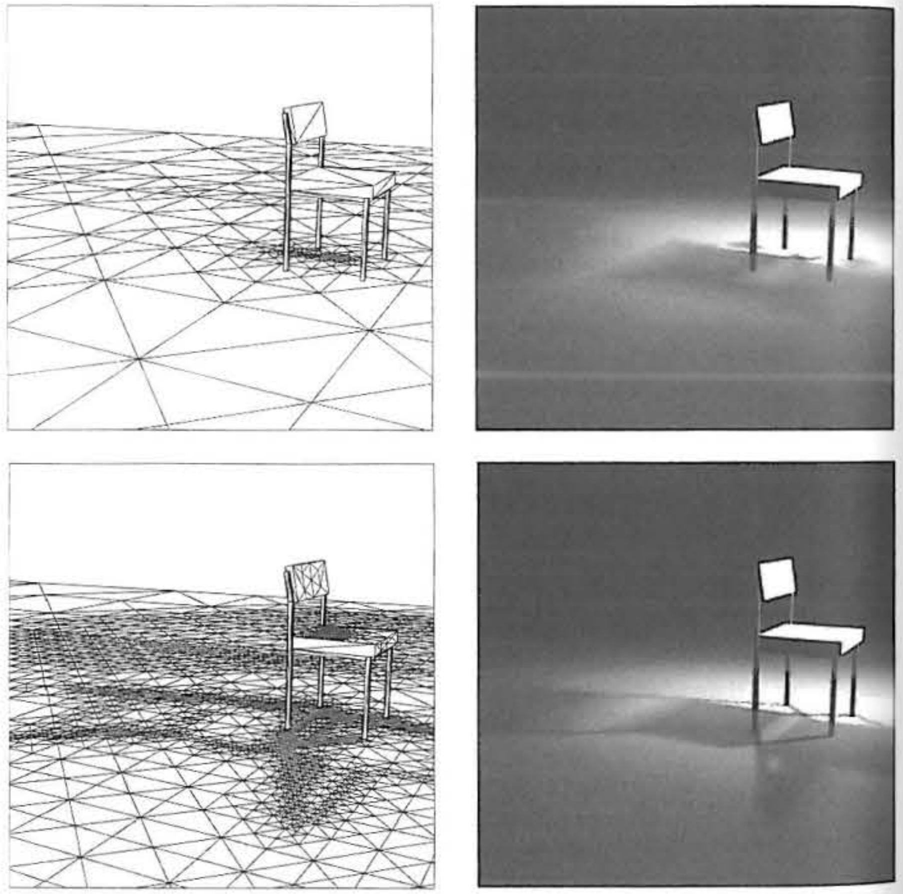
(1) Subdividing selected patches into elements and calculating element-to-patch form factors.
(2) Evaluating a radiosity solution using patch-to-patch form factors.
(3) Determining the element radiosities from the patch radiosities.

Where stage (2) just occurs for the first iteration, the coarse patch radiosities are calculated once only. The method is distinguished from simply subdividing the environment into smaller patches. This strategy would result in $M \times M$ new form factors (rather than $M \times N$) and an $M \times M$ system of equations.

Subdivision of patches into elements is carried out adaptively. The areas that require subdivision are not known prior to a solution being obtained. These areas are obtained from an initial solution and are then subject to a form factor subdivision. The previous form factor matrix is still valid and the radiosity solution is not re-computed.

Only part of the form factor determination is further discretized and this is then used in the third phase (determination of the element radiosities from the coarse patch solution). This process is repeated until it converges to the desired degree of accuracy. Thus image quality is improved in areas that require more accurate treatment. An example of this approach is shown in Figure 11.13. Note the effect on the quality of the shadow boundary. Figure 11.14 shows the same set-up but this time the light source is subdivided to a lower and higher resolution than in Figure 11.13. Although the effect, in this case, of insufficient subdivision of emitting and non-emitting patches is visually similar, the reasons for these discrepancies differ. In the case of non-emitting patches we have changes in reflected light intensity that do not coincide with patch boundaries. We increase the number of patches to capture the discontinuity. With emitting patches the problem is due to the number of hemicube emplacements per light source. Here we increase the number of patches that represent the emitter because each hemicube emplacement reduces a light to a single source and we need a sufficiently dense array of these to represent the spatial extent of the

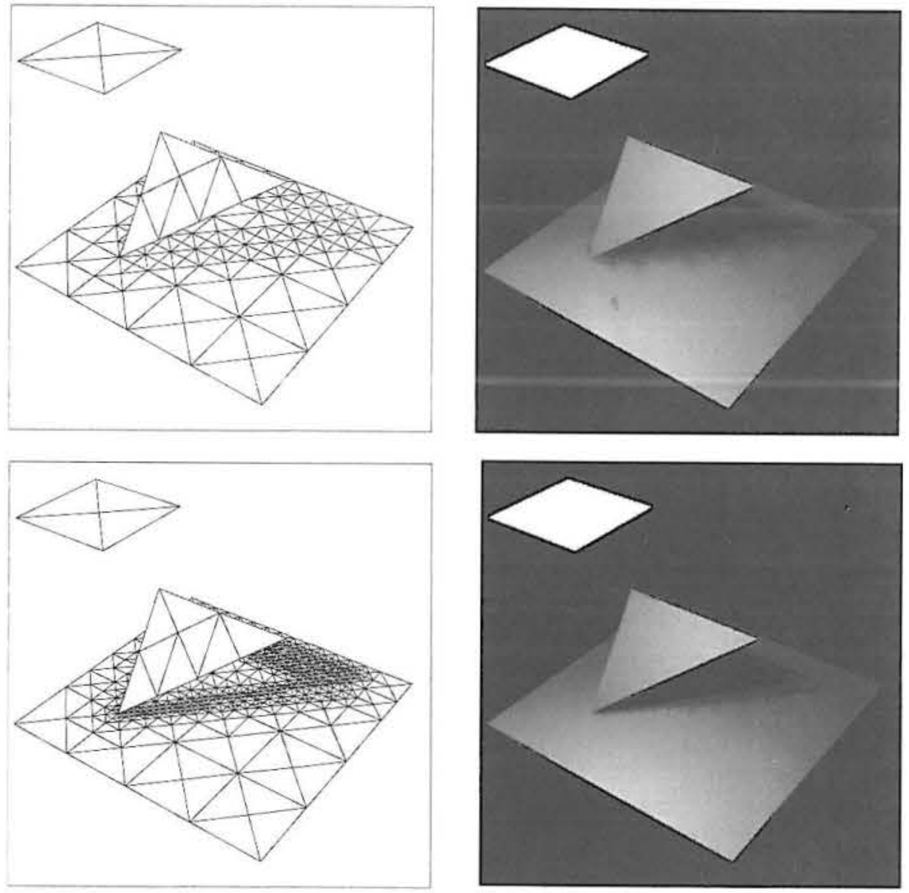
Figure 11.13
Adaptive subdivision and shadows.


(a) Shape and shadow areas do not correspond to shape of the occluder.

emitter. In this case we are subdividing a patch (the emitter) over whose surface the light intensity will be considered uniform.

Adaptive subdivision can be incorporated in the progressive refinement method. A naive approach would be to compute the radiosity gradient and subdivide based on the contribution of the current shooting patch. However, this approach can lead to unnecessary subdivisions. The sequence, shown in Figure 11.15 shows the difficulties encountered as subdivision, performed after every iteration, proceeds around one of the wall lights. Originally two large patches situated away from the wall provide general illumination of the object. This immediately causes subdivision around the light–wall boundary because the program detects a high difference between vertices belonging to the same patches. These patches have vertices both under the light and on the wall. However, this subdivision is not fine enough and as we start to shoot energy from the light source itself light leakage begins to occur. Light source patches continue to shoot energy in the order in which the model is stored in the data-

Figure 11.13 *continued*


(b) Boundary of shadow is jagged.

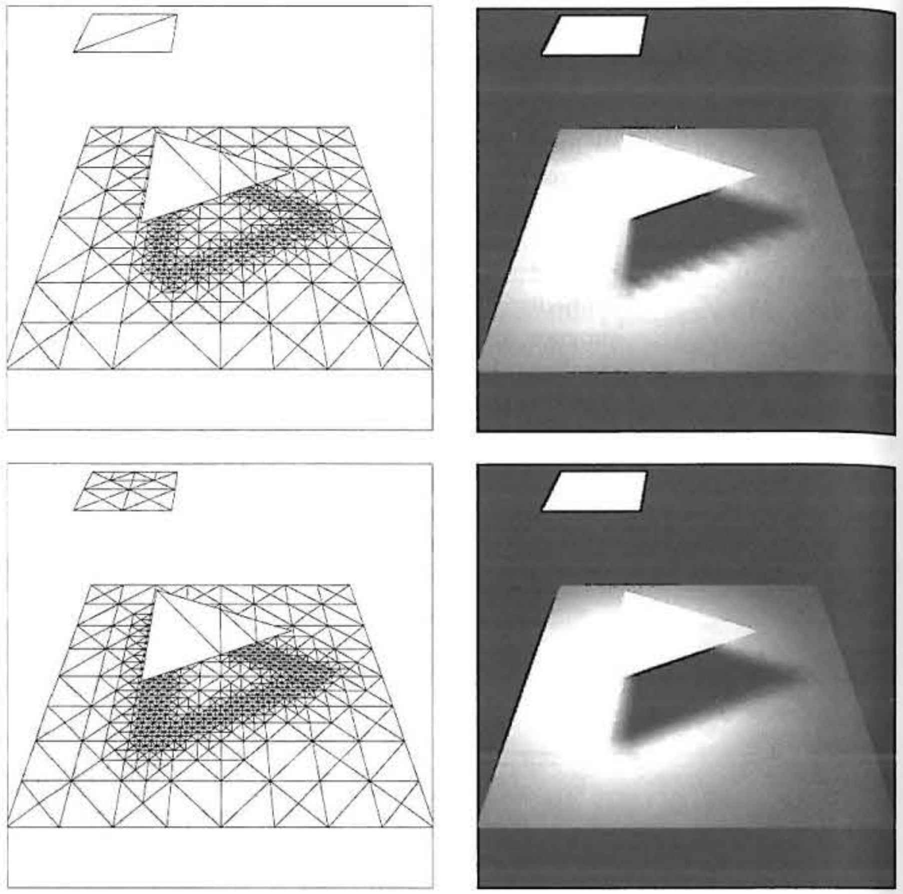


Figure 11.14
The effect of insufficient subdivision of emitters.

base and we spiral up the sphere, shooting energy onto its inside and causing more and more light leakage. Eventually the light emerges onto the wall and brightens up the appropriate patches. As the fan of light rotates above the light more and more inappropriate subdivision occurs. This is because the subdivision is based on the current intensity gradients which move on as further patches are shot. Note in the final frame this results in a large degree of subdivision in an area of highlight saturation. These redundant patches slow the solution down more and more and we are inadvertently making things worse as far as execution time is concerned.

Possible alternative strategies are:

(1) Limit the subdivision by only initiating it after every n patches instead of after every patch that is shot.
(2) Limit the initiation of subdivision by waiting until the illumination is representative of the expected final distribution.

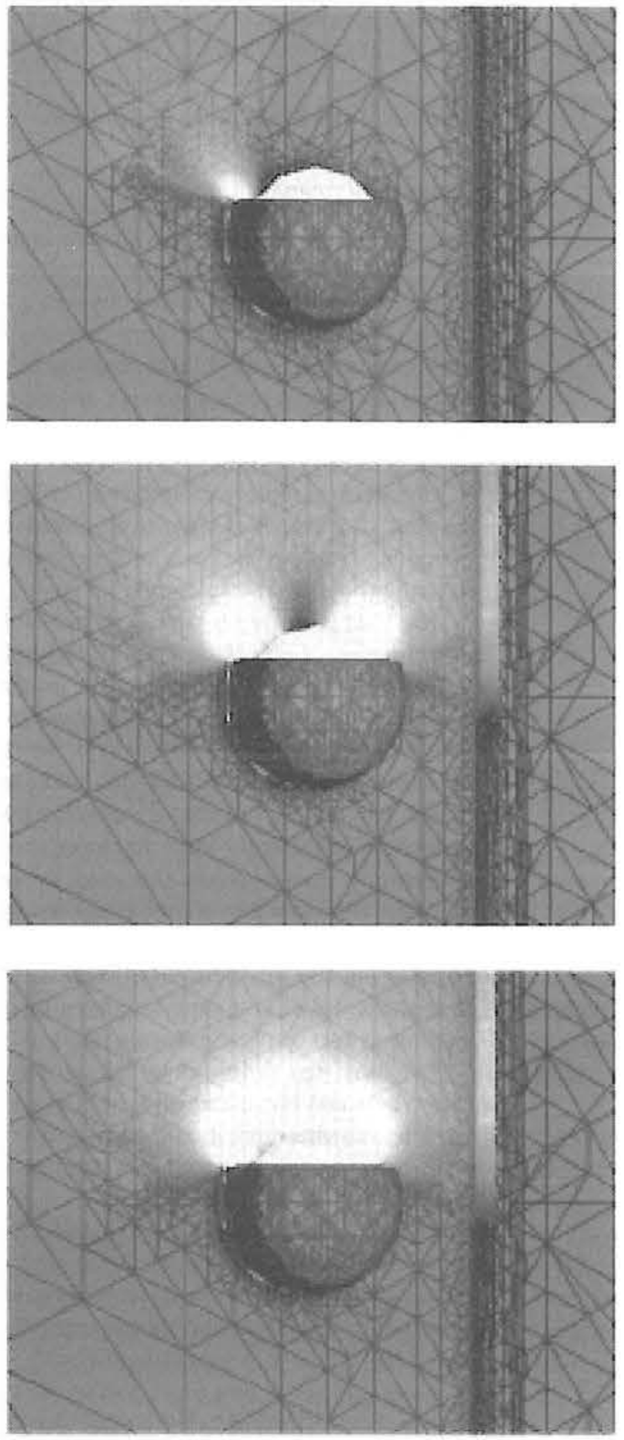


Figure 11.15
The sequence shows the difficulties encountered as subdivision, performed after every iteration, proceeds around one of the wall lights. As the fan of light rotates above the light more and more inappropriate subdivision occurs. This is because the subdivision is based on the current intensity gradients which move on as further patches are shot. Note in the final frame this results in a large degree of subdivision in an area of high light saturation. These redundant patches slow the solution down more and more and we are inadvertently making things worse as far as execution time is concerned.

11.7.2 *A priori* meshing

We will now look at two strategies for *a priori* meshing – processing a scene before it is used in a radiosity solution.

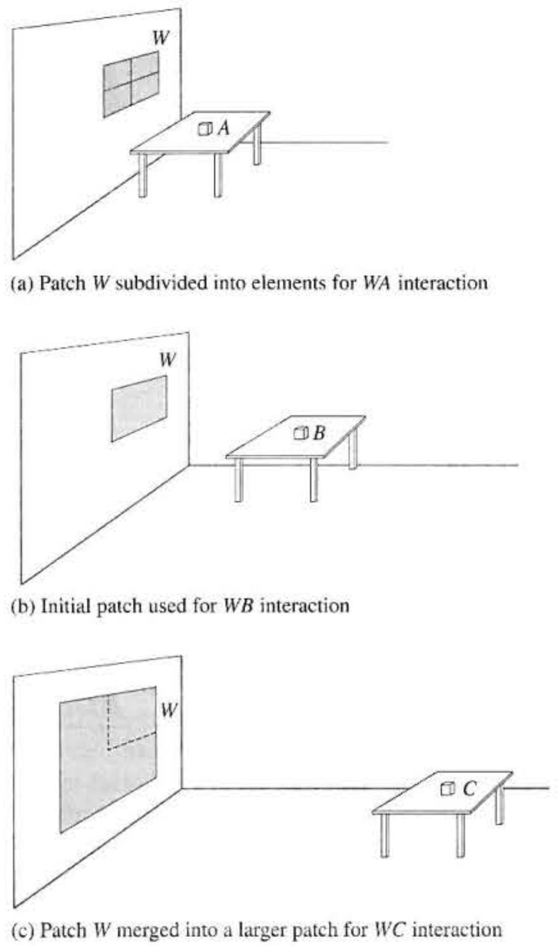
Hierarchical radiosity

Adaptive subdivision is a special case of an approach which is nowadays known as hierarchical radiosity. Hierarchical radiosity is a generalization of adaptive subdivision – a two-level hierarchy – to a subdivision employing a continuum of levels. The interaction between surfaces is then computed using a form factor appropriate to the geometric relationship between the two surfaces. In other words, the hierarchical approach attempts to limit form factor calculation time by limiting the accuracy of the calculation to an amount determined by the distance between patches.

Hierarchical radiosity can be embedded in a progressive refinement approach making an *a posteriori* algorithm; alternatively the hierarchical subdivision can be made on an *a priori* basis and the system then solved. In what follows we will describe the *a priori* framework.

The idea is easily illustrated in principle. Figure 11.16 shows a wall patch W and three small objects A, B and C located at varying distances from W. The distance from W to A is comparable to its dimension and we assume that W has to be subdivided to calculate the changes in illumination in the vicinity of A due to light emitted or reflected from W. In the case of B we assume that the whole of patch W can be used. Detailed variation of the radiosity in the vicinity of B due to patch W is not unduly affected by subdividing B. The distance to C, we assume, is sufficiently large to make the form factor between W and C correspondingly small, and in this case we can consider a larger area on the wall merging W into a patch four times its area. Thus for the three interactions we use either a subdivided W, the whole of W or W as part of a larger entity when considering the interaction between the wall and the objects A, B and C. Note that this implies not only subdivision of patches but the opposite process – agglomeration of patches into groups.

The idea, first proposed by Hanrahan *et al.* (1991), proposes that if the form factor between two patches currently under consideration exceeds a threshold, then to use these patches at their current size will introduce an unacceptable error into the solution and the patches should be subdivided. Compared with the strategy in the previous section we are taking our differential threshold one stage further back in the overall process. Instead of comparing the difference between the calculated radiosity of neighbouring patches and subdividing and re-calculating form factors if necessary, we are looking directly at the form factors themselves and subdividing until the form factor falls below a threshold. This idea is easily demonstrated for the simple case of two patches sharing a common border. Figure 11.17 shows the geometric effect of subdivision based on a form factor threshold. The initial form factor estimate is large and the patches



(a) Patch W subdivided into elements for WA interaction

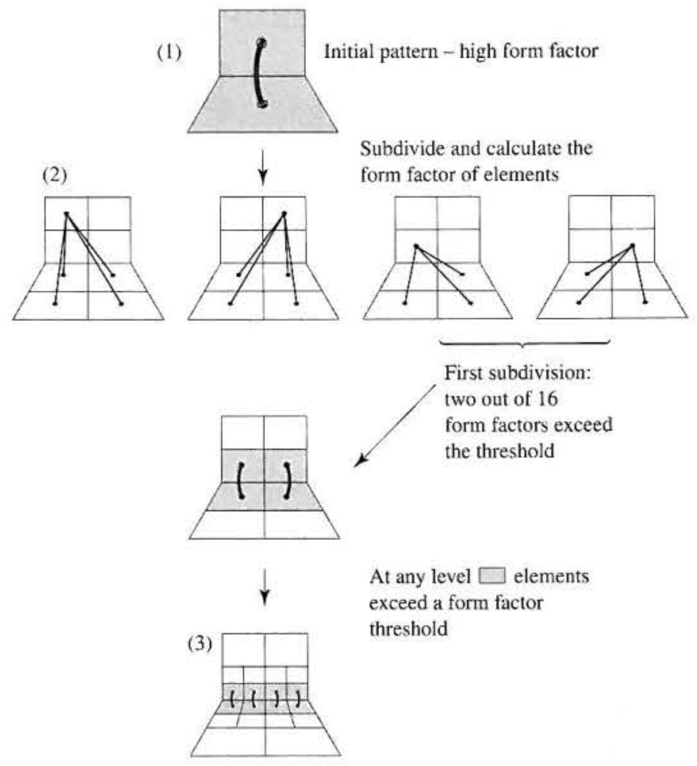
(b) Initial patch used for WB interaction

(c) Patch W merged into a larger patch for WC interaction

Figure 11.16 Hierarchical radiosity: scene subdivision is determined by energy interchange.

are subdivided into four elements. At the next level of subdivision only two out of 16 form factor estimates exceed the threshold and they are subdivided. It is easily seen from the illustration that in this example the pattern of subdivision 'homes into' the common edge.

A hierarchical subdivision strategy starts with an (initial) large patch subdivision of n patches. This results in $n(n-1)/2$ form factor calculations. Pairs of patches that cannot be used at this initial level are then subdivided as suggested by the previous figure, the process continuing recursively. Thus each initial patch is represented by a hierarchy and links. The structure contains both the geometric subdivision and links that tie an element to other elements in the scene. A node in the hierarchy represents a group of elements and a leaf node a single element. To make this process as fast as possible a crude estimate of the form factor can be used. For example, the expression inside the integral definition of the form factor:



(1)
Initial pattern – high form factor
(2)
Subdivide and calculate the form factor of elements
First subdivision: two out of 16 form factors exceed the threshold
At any level ▭ elements exceed a form factor threshold
(3)

Figure 11.17
Hierarchical radiosity: the geometric effect of subdivision of two perpendicular patches (after Hanrahan *et al.* (1991)).

$$\frac{\cos \phi_i \cos \phi_j}{\pi r^2}$$

can be used but note that this does not take into account any occluding patches.

Thus the stages in establishing the hierarchy are:

(1) Start with an initial patch subdivision. This would normally use much larger patches than those required for a conventional solution.

(2) Recursively apply the following:
   (a) Use a quick estimate of the form factor between pairs of linked surfaces.
   (b) If this falls below a threshold or a subdivision limit is reached, record their interaction at that level.
   (c) Subdivide the surfaces.

It is important to realize that two patches can exhibit an interaction between any pair of nodes at any level in their respective hierarchies. Thus in Figure 11.18 a link is shown between a leaf node in patch *A* and an internal node in patch *C*. The tree shown in the figure for *A* represents the subdivisions necessary for its interaction with patch *B* and that for patch *C* represents its interactions with some other patch *X*. This means that energy transferred from *A* to the internal node in *C* is inherited by all the child nodes below the destination in *C*.

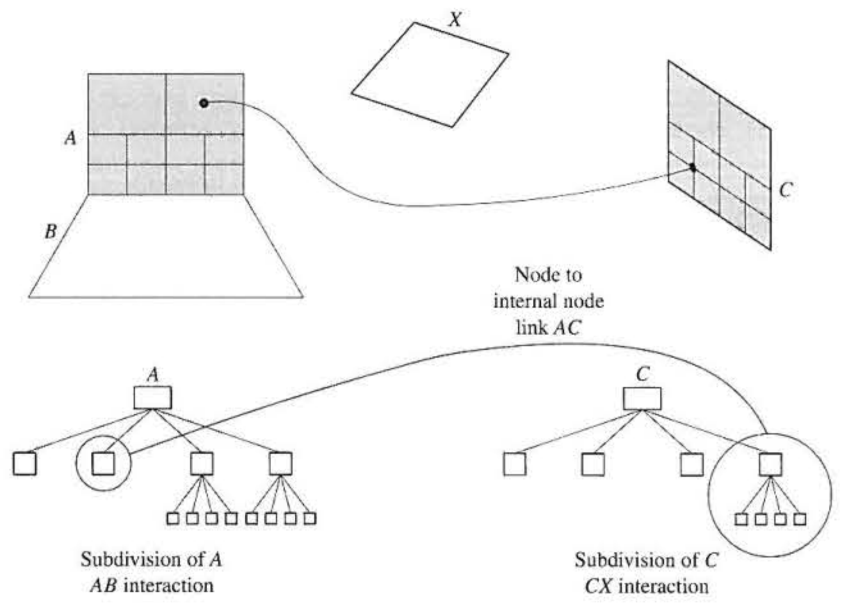


Figure 11.18
Interaction between two patches can consist of energy interchange between any pair of nodes at any level in their respective hierarchy.

Comparing this formulation with the classical full matrix solution we have now replaced the form factor matrix with a hierarchical representation. This implies that a 'gathering' solution proceeds from each node by following all the links from that node and multiplying the radiosity found at the end of the link by the associated form factor. Because links have, in general, been established at any level, the hierarchy needs to be followed in both directions from a node.

The iterative solution proceeds by executing two steps on each root node until a solution has converged. The first step is to gather energy over each incoming link. The second step, known as 'pushpull' pushes a node's reflected radiosity down the tree and pulls it back up again. Pushing involves simply adding the radiosity at each child node. (Note that since radiosity has units of power/unit area the value remains undiminished as the area is subdivided.) The total energy received by an element is the sum of the energy received by it directly plus the sum of the energy received by its parents. When the leaves are reached the process is reversed and the energy is pulled up the tree, the current energy deposited at each node is calculated by averaging the child node contributions.

In effect this is just an elaboration of the Gauss–Siedel relaxation method described in Section 11.3. For a particular patch we are gathering contributions from all other patches in the scene to enable a new estimate of the current patch. The difference now is that the gathering process involves following all links out of the current hierarchy and the hierarchy has to be updated correctly with the bi-directional traversal or pushpull process.

The above algorithm description implies that at each node in the quadtree data structure the following information is available:

gathering and shooting radiosity (B_g and B_s)
emission value (E)
area (A)
reflectivity (ρ)
pointer to four children
pointer to list of (gathering) links (L)

The algorithm itself has a simple elegant structure and following the excellent treatment given in Cohen and Wallace (1993) can be expressed in terms of the following pseudocode:

```
while not converged
    for all surfaces (every root node p)
            GatherRad(p)
    for all surfaces
            PushPull(p,0.0)
```

The top level procedure is a straightforward iterative process which gathers all the energy from the incoming links, pushes it down the structure then pulls the radiosity values back up the hierarchy.

GatherRad(p) calculates the radiosity absorbed and then reflected at node p. It is as follows:

```
GatherRad(p)
    $p.B_g := 0$
    for each link L into p
    $p.B_g := p.B_g + p.\rho(L.F_{pq} * L.q.B_s)$
    for each child r of p
      GatherRad(r)
```

Here q is the element linked to p by L, F_{pq} is the form factor for this link and ρ is the reflectivity of the element p. The radiosity at the destination/shooter end of the link is B_s. This is converted into the reflected radiosity B_g at the source/gatherer end of the link by multiplying it by the form factor F_{pq} and the reflectivity ρ.

PushPullRad(p,B) can be viewed as a procedure that distributes the energy correctly throughout the hierarchy balancing the tree. It is as follows:

```
PushPullRad(p,$B_{down}$)
    if p is a leaf then $B_{up} := p.E + p.B_g + B_{down}$
    else $B_{up} := 0$; for each child node r of p
                    $B_{up} := B_{up} + (r.A/p.A)$ * PushPullRad(r, $p.B_g + B_{down}$)
    $p.B_s := B_{up}$
    return $B_{up}$
```

The procedure is first called at the top of the hierarchy with the gathered radiosity at that level. The recursion has the effect of passing or pushing down this radiosity onto the child nodes. At each internal node the gathered power is added to the inherited power accumulated along the downwards path. When a leaf node is reached any emission is added into the gathered radiosity for that node and the result assigned to the shooting radiosity for that node. The recursion then unwinds pulling the leaf node radiosity up the tree and performing an area weighting at each node.

Although hierarchical radiosity is an efficient method and one that can be finely controlled (the accuracy of the solution depends on the form factor tolerance and the minimum subdivision area) it still suffers from shadow leaks and jagged shadow boundaries because it subdivides the environment regularly (albeit non-uniformly) without regard to the position of shadow boundaries. Reducing the value of the control parameters to give a more accurate solution can still be prohibitively expensive. This is the motivation of the approach described in the next section.

Finally we can do no better than to quote from the original paper, in which the authors give their inspiration for the approach:

The hierarchical subdivision algorithm proposed in this paper is inspired by methods recently developed for solving the N-body problem. In the N-body problem, each of the n particles exerts a force on all the other $n-1$ particles, implying $n(n-1)/2$ pairwise interactions. The fast algorithm computes all the forces on a particle in less than quadratic time, building on two key ideas:

(1) Numerical calculations are subject to error, and therefore, the force acting on a particle need only be calculated to within the given precision.

(2) The force due to a cluster of particles at some distant point can be approximated, within the given precision, with a single term – cutting down on the total number of interactions.

Discontinuity meshing

The commonest, and simplest, type of *a priori* meshing is to take care of the special case of interpenetrating geometry (D^0) as we suggested at the beginning of this section. This is mostly done semi-manually when the scene is constructed and disposes of shadow and light leakage – the most visible radiosity artefact. The more general approaches attend to higher-order discontinuities. D^1 and D^2 discontinuities occur when an object interacts with an area light source – the characteristic penumbra–umbra transition within a shadow area – as described in Chapter 9.

As we have seen, common *a posteriori* methods generally approach the problem by subdividing in the region of discontinuities in the radiosity function and can only eliminate errors by resorting to higher and higher meshing densities. The idea behind discontinuity meshing is to predict where the discontinuities are going to occur and to align the mesh edges exactly with the path of the discontinuity. This approach is by definition an *a priori* method. We predict where the discontinuities will occur and mesh, before invoking the solution phase so that when the solution proceeds there can be no artefacts present due to the non-alignment of discontinuities and mesh edges.

To predict the position of the discontinuities shadow detection algorithms are used and the problem is usually couched in terms of visual events and critical surfaces. Two types of visual events can be considered VE and EEE. VE or vertex–edge events occur when a vertex of a source 'crosses' an edge of an occluding polygon known in this context as a receiver. Figure 11.19 shows the interaction between a vertex of a triangular source and an edge of a rectangular occlude. The edge and vertex together form a critical surface whose intersection with a receiving surface forms part of the outer penumbra boundary. For each edge of the occluder a critical surface can be defined with respect to each vertex in the source. We can also define EV events which occur due to the interaction of a source edge with a receiver polygon.

VE events can cause both D^1 and D^2 discontinuities as Figures 11.20 and 11.21 demonstrate. Figure 11.20 shows the case of a D^1 discontinuity. Here there is the coincidence that the edge of the occluder and the source are parallel. Both vertices V_1 and V_2 contribute to the penumbra. As we travel outwards from the umbra along path *xy*, the visible area of the source increases linearly and the radiance exhibits piecewise linearity or D^1 discontinuities. A D^2 discontinuity caused by a VE event is shown in Figure 11.21. In this case, a single vertex of the light source is involved along the path *xy*. As we travel outwards from the umbra the visible area of the source increases quadratically and the radiance exhibits D^2 discontinuities.

EEE or edge–edge–edge events occur when we have multiple occluders. The important difference here is that the boundary of the penumbra – the critical curve – is no longer a straight line as it was in the previous VE examples but a conic. The corresponding discontinuities in the radiance function along the curve are D^2. Also the critical surface is no longer a segment of a plane but is a (ruled) quadric surface.

Visual events can occur for any edges and vertices of any object in the scene. For a scene with n objects there can be $O(n^2)$ VE critical surfaces and $O(n^3)$ EEE critical surfaces. Because of the cost and the higher complexity of EEE events approaches have concentrated on detecting VE events.

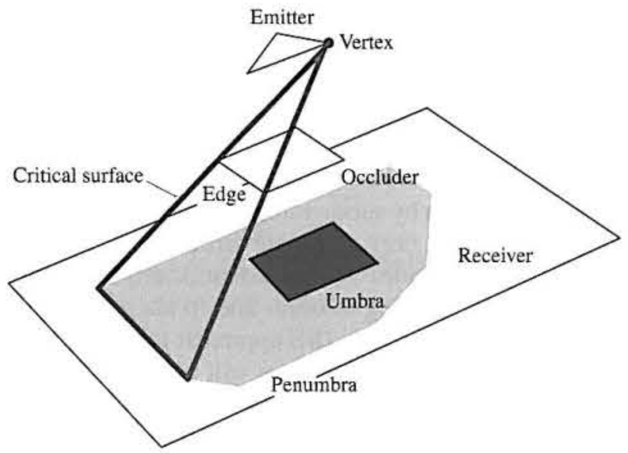


Figure 11.19
VE event: the edge of the occluder and a vertex of the emitter form a critical surface whose intersection with the receiver forms the outer boundary of the penumbra (after Nishita and Nakamae (1985)).

Figure 11.20
VE event causing a D^1 discontinuity (after Lischinski *et al.* (1992)).

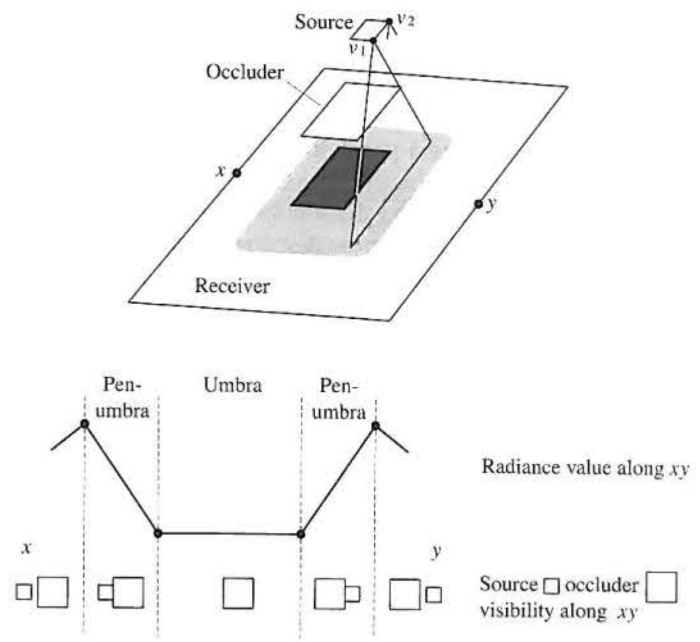
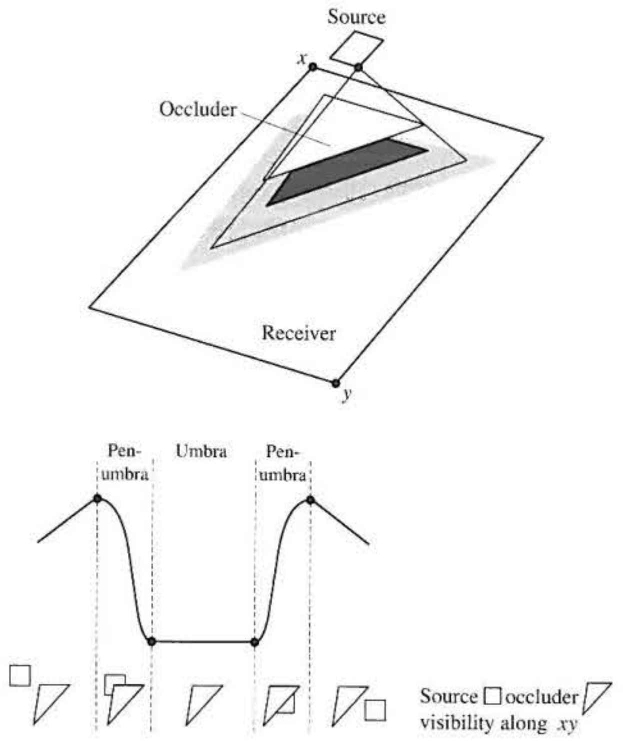


Figure 11.21
VE event causing a D^2 discontinuity (after Lischinski *et al.* (1992)).



A straightforward approach by Nishita and Nakamae (1985) explicitly determines penumbra and umbra boundaries by using a shadow volume approach. For each object a shadow volume is constructed from each vertex in the light source. Thus, there is a volume associated with each light source vertex just as if it were a point source. The intersection of all volumes on the receiving surface forms the umbra and the penumbra boundary is given by the convex hull containing the shadow volumes. An example is shown in Figure 11.22.

We will now describe in some detail a later and more elaborate approach by Lischinski *et al.* (1992) to discontinuity meshing. This integrates discontinuity meshing into a modified progressive refinement structure and deals only with VE (and EV) events. This particular algorithm is representative in that it deals with most of the factors that must be addressed in a practical discontinuity meshing approach including handling multiple light sources and reconstruction problems.

Lischinski *et al.* build a separate discontinuity mesh for each source, accumulating the results into a final solution. The scene polygons are stored as a BSP tree which means that they can be fetched in front-to-back order from a source vertex. For a source the discontinuities that are due to single VE events are located as follows. Figure 11.23 shows a single VE event generating a wedge defined by the vertex and projectors through the end points of the edge, *E*. The event is processed by fetching the polygons in the order *A*, *B* and *C*. *A* is nearer to the source than *E* and is thus not affected by the event. If a surface (*B* and *C*) faces the source then the intersection of the wedge with the surface adds a

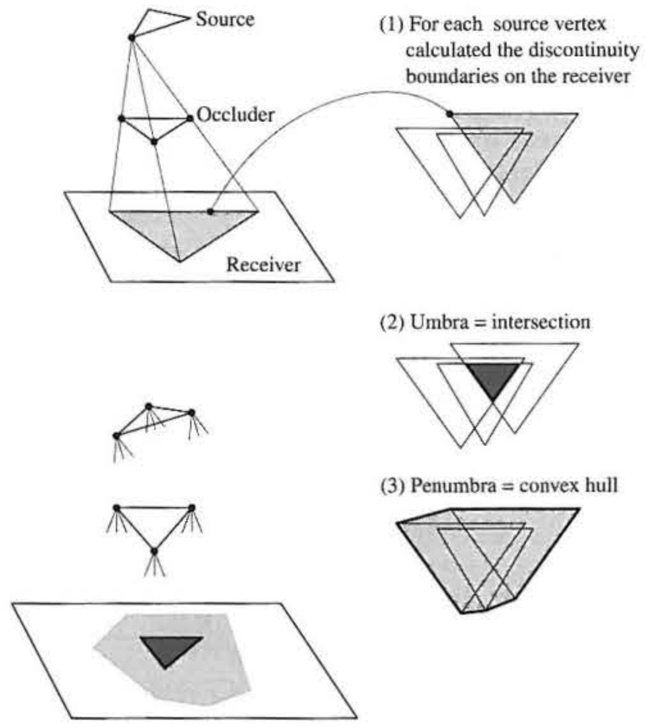
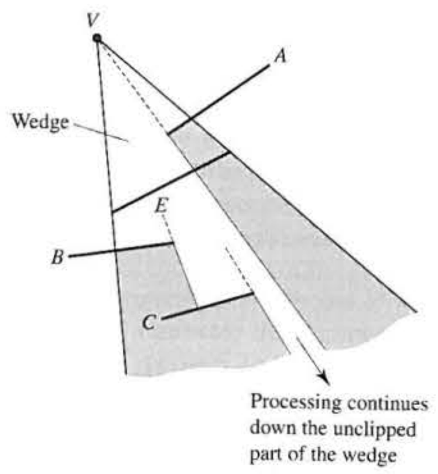

Source
Occluder
Receiver
(1) For each source vertex calculated the discontinuity boundaries on the receiver
(2) Umbra = intersection
(3) Penumbra = convex hull

Figure 11.22
Umbra and penumbra from shadow volumes formed by VE events.

Figure 11.23
Processing a VE wedge (after Lischinski *et al.* (1992)).



discontinuity to that surface. The discontinuity is 'inserted' into the mesh of the surface. As each surface is processed it clips out part of the wedge and the algorithm proceeds 'down' only the unclipped part of the wedge. When the wedge is completely clipped the processing of that particular VE event is complete.

The insertion of the discontinuity into the mesh representing the surface is accomplished by using a DM tree which itself consists of two components – a two-dimensional BSP tree connecting into a winged edge data structure (Mantyla 1988) representing the interconnection of surface nodes. The way in which this works is shown in Figure 11.24 for a single example of a vertex generating three VE events which plant three discontinuity/critical curves on a receiving surface. If the processing order is *a*, *b*, *c* then the line equation for *a* appears as the root node and splitting it into two regions as shown. *b*, the next wedge to be processed is checked against region R_1 which splits into R_{11} and R_{12} and so on.

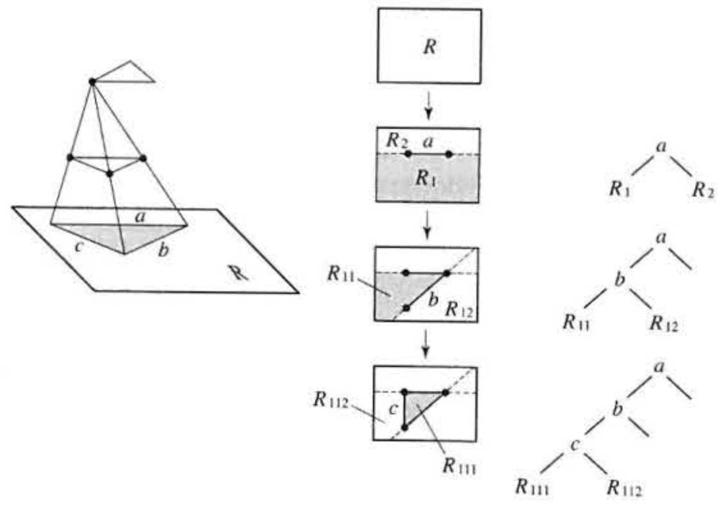


Figure 11.24
Constructing a DM tree for a single VE event (after Lischinski *et al.* (1992)).

12 Ray tracing strategies

Introduction – Whitted ray tracing

In Chapter 10 we gave a brief overview of Whitted ray tracing. We will now describe this popular algorithm in detail. Although it was first proposed in by Appel (1968), ray tracing is normally associated with Whitted's classic paper (1980). We use the term 'Whitted ray tracing' to avoid the confusion that has arisen due to the proliferation of adjectives such as 'eye', forward' and 'backward' to describe ray tracing algorithms.

Whitted ray tracing is an elegant partial global illumination algorithm that combines the following in a single model:

- Hidden surface removal.
- Shading due to direct illumination.
- Global specular interaction effects such as the reflection of objects in each other and refraction of light through transparent objects.
- Shadow computation (but only the geometry of hard-edged shadows is calculated).

It usually 'contains' a local reflection model such as the Phong reflection model, and the question arises: why not use ray tracing as the standard approach to rendering, rather than using a Phong approach with extra algorithms for hidden

surface removal, shadows and transparency? The immediate answer to this is cost. Ray tracing is expensive, particularly for polygon objects because effectively each polygon is an object to be ray traced. This is the dilemma of ray tracing. It can only function in reasonable time if the scene is made up of 'easy' objects. Quadric objects, such as spheres are easy, and if the object is polygonal the number of facets needs to be low for the ray tracer to function in reasonable time. If the scene is complex (many objects each with many polygons) then the basic algorithm needs to be burdened with efficiency schemes whose own cost tends to be a function of the complexity of the scene. Much of the research into ray tracing in the 1980s concentrated on the efficiency issue. However, we are just about at a point in hardware development where ray tracing is a viable alternative for a practical renderer and the clear advantages of the algorithm are beginning to overtake the cost penalties. In this chapter we will develop a program to ray trace spheres. We will then extend the program to enable polygonal objects to be dealt with.

12.1 The basic algorithm

12.1.1 Tracing rays – initial considerations

We have already seen that we trace infinitesimally thin light rays through the scene, following each ray to discover perfect specular interactions. Tracing implies testing the current ray against objects in the scene – intersection testing – to find if the ray hits any of them. And, of course, this is the source of the cost in ray tracing – in a naive algorithm, for each ray we have to test it against all objects in the scene (and all polygons in each object). At each boundary, between air and an object (or between an object and air) a ray will 'spawn' two more rays. For example, a ray initially striking a partially transparent sphere will generate at least four rays for the object – two emerging rays and two internal rays (Figure 10.5). The fact that we appropriately bend the transmitted ray means that geometric distortion due to refraction is taken into account. That is, when we form a projected image, objects that are behind transparent objects are appropriately distorted. If the sphere is hollow the situation is more complicated – there are now four intersections encountered by a ray travelling through the object.

To perform this tracing we follow light beams in the reverse direction of light propagation – we trace light rays from the eye. We do this eye tracing because tracing rays by starting at the light source(s) would be hopelessly expensive. This is because we are only interested in that small subset of light rays which pass through the image plane window.

At each hit point the same calculations have to be made and this implies that the easiest way to implement a simple ray tracer is as a recursive procedure. The recursion can terminate according to a number of criteria:

- It always terminates if a ray intersects a diffuse surface.
- It can terminate when a pre-set depth of trace has been reached.
- It can terminate when the energy of the ray has dropped below a threshold.

The behaviour of such an approach is demonstrated in Figure 18.11 (Colour Plate). Here the trace is terminated at recursives depths of 2, 3 and 4 and unassigned pixels (pixels which correspond to a ray landing on a pure specular surface with no diffuse contribution) are coloured grey. You can see that the grey region 'shrinks into itself' as a function of recursive depth.

12.1.2 Lighting model components

At each point P that a ray hits an object, we spawn in general, a reflected and a transmitted ray. Also we evaluate a local reflection model by calculating L at that point by shooting a ray to the light source which we consider as a point. Thus at each point the intensity of the light consists of up to three components:

- A local component.
- A contribution from a global reflected ray that we follow.
- A contribution from a global transmitted ray that we follow.

We linearly combine or add these components together to produce an intensity for point P. It is necessary to include a local model because there may be direct illumination at a hit point. However, it does lead to this confusion. The use of a local reflection model does imply empirically blurred reflection (spread highlights); however, the global reflected ray at that point is not blurred but continues to discover any object interaction along an infinitesimally thin path. This is because we cannot afford to blur global reflected rays – we can only follow the 'central' ray. This results in a visual contradiction in ray traced images, which is that the reflection of the light source in an object – the specular highlight – is blurred, but the images of other objects are perfect. The reason for this is that we want objects to look shiny – by having them exhibit a specular highlight – and include images of other objects. Thus most algorithms use a local and a global specular component.

It is also necessary to account for local diffuse reflection, otherwise we could not have coloured objects. We cannot in ray tracing handle diffuse interaction as we did in radiosity. This would mean spawning, for every hit, a set of diffuse rays that sampled the hemispherical set of diffuse rays that occurs at the hit point on the surface of the object, if it happens to be diffuse. Each one of these rays would have to be followed and may end up on a diffuse surface and a combinatorial explosion would develop that no machine could cope with. This problem is the motivation for the development of Monte Carlo methods such as path tracing, as we saw in Chapter 10.

If a ray hits a pure diffuse surface then the trace is terminated. Thus we have the situation where the result of the local model computation at each hit point is passed up the tree along with the specular interaction.

12.1.3 Shadows

Shadows are easily included in the basic ray tracing algorithm. We simply calculate L, the light direction vector, and insert it into the intersection test part of the algorithm. That is, L is considered a ray like any other. If L intersects any objects, then the point from which L emanates is in shadow and the intensity of direct illumination at that point is consequently reduced (Figure 12.1). This generates hard-edged shadows with arbitrary intensity. The approach can also lead to great expense. If there are n light sources, then we have to generate n intersection tests. We are already spawning two rays per hit point plus a shadow ray, and for n light sources this becomes $(n + 2)$ rays. We can see that as the number of light sources increases shadow computations are quickly going to predominate since the major cost at each hit point is the cost of the intersection testing.

In an approach by Haines and Greenberg (1986) a 'light buffer' was used as a shadow testing accelerator. Shadow testing times were reduced, using this procedure, by a factor of between 4 and 30. The method pre-calculates for each light source, a light buffer which is a set of cells or records, geometrically disposed as two-dimensional arrays on the six faces of a cube surrounding a point light source (Figure 12.2). To set up this data structure all polygons in the scene are cast or projected onto each face of the cube, using as a projection centre the position of the light source. Each cell in the light buffer then contains a list of polygons that can be seen from the light source. The depth of each polygon is calculated in a local coordinate system based on the light source, and the records are sorted in ascending order of depth. This means that for a particular ray from the eye, there is immediately available a list of those object faces that may occlude the intersection point under consideration.

Shadow testing reduces to finding the cell through which the shadow feeler ray passes, accessing the list of sorted polygons, and testing the polygons in the list until occlusion is found, or the depth of the potentially occluding polygon is greater than that of the intersection point (which means that there is no occlusion because the polygons are sorted in depth order). Storage requirements are prodigious and depend on the number of light sources and the resolution of the

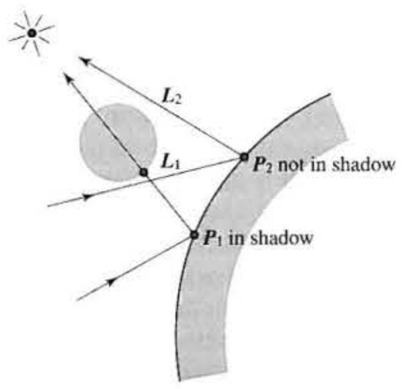


Figure 12.1
Shadow shape is computed by calculating L and inserting it into the intersection tester.

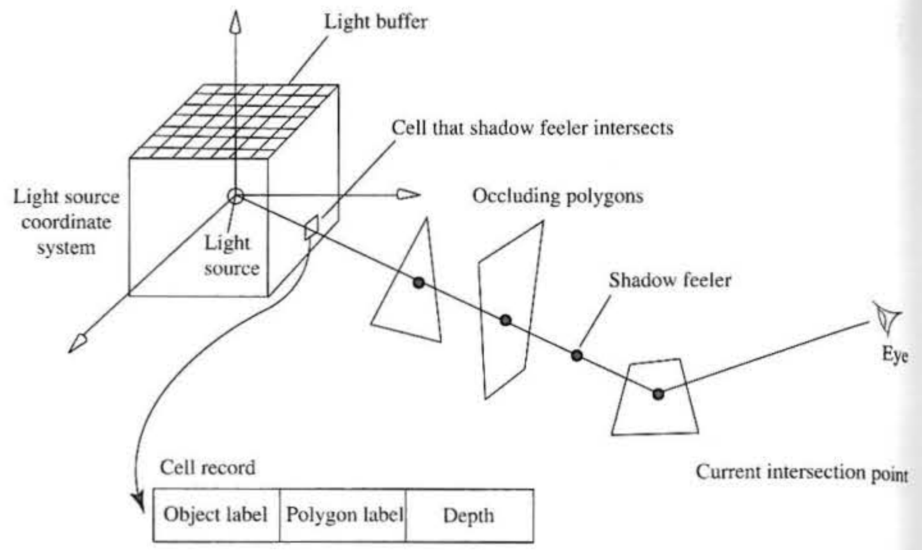


Figure 12.2
Shadow testing accelerator of Haines and Greenberg (1986).

light buffers. (Note the similarity between the light buffer and the radiosity hemicube described in Chapter 11.)

Apart from the efficiency consideration, the main problem with shadows in Whitted ray tracing is that they are hard edged due to the point light source assumption and the light intensity within a shadow area has to be a guess. This is, of course, not inconsistent with the perfect specular interactions that result from tracing a single infinitesimally thin ray from each hit point for each type of interaction. Just as distributed ray tracing (described in Section 10.6) deals with 'blurry' interaction by considering more than one ray per interaction, so it implements soft shadow by firing more than one ray towards a (non-point) light source.

12.1.4 Hidden surface removal

Hidden surface removal is 'automatically' included in the basic ray tracing algorithm. We test each ray against all objects in the scene for intersection. In general this will give us a list of objects which the ray intersects. Usually the intersection test will reveal the distance from the hit point to the intersection and it is simply a matter of looking for the closest hit to find, from all the intersections, the surface that is visible from the ray-initiating view point. A certain subtlety occurs with this model, which is that surfaces hidden, from the point of view of a standard rendering or hidden surface approach, may be visible in ray tracing. This point is illustrated in Figure 12.3 which shows that a surface, hidden when viewed from the eye ray direction, can be reflected in the object hit by the incident ray.

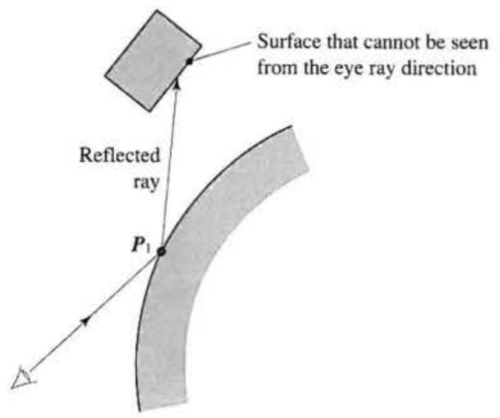


Figure 12.3
A reflected 'hidden' surface.

12.2 Using recursion to implement ray tracing

We will now examine the working of a ray tracing algorithm using a particular example. The example is based on a famous image, produced by Turner Whitted in 1980, and it is generally acknowledged as the first ray traced image in computer graphics. An imitation is shown in Figure 12.4 (reproduced as a monochrome image here and colour image in the Colour Plate section).

First some symbolics. At every point $\boldsymbol{P}$ that we hit with a ray we consider two major components a local and a global component:

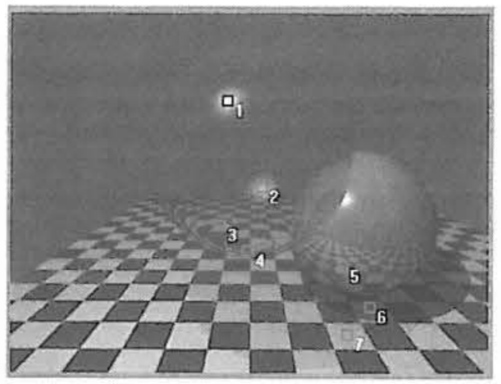
$$\begin{aligned} I(\boldsymbol{P}) &= I_{\text{local}}(\boldsymbol{P}) + I_{\text{global}}(\boldsymbol{P}) \\ &= I_{\text{local}}(\boldsymbol{P}) + k_{\text{rg}}\, I(\boldsymbol{P}_{\text{r}}) + k_{\text{tg}} I(\boldsymbol{P}_{\text{t}}) \end{aligned}$$

where:

$\boldsymbol{P}$ is the hit point
$\boldsymbol{P}_{\text{r}}$ is the hit point discovered by tracing the reflected ray from $\boldsymbol{P}$
$\boldsymbol{P}_{\text{t}}$ is the hit point discovered by tracing the transmitted ray from $\boldsymbol{P}$
k_{rg} is the global reflection coefficient
k_{tg} is the global transmitted coefficient

This recursive equation emphasizes that the illumination at a point is made up of three components, a local component, which is usually calculated using a Phong local reflection model, and a global component, which is evaluated by finding $\boldsymbol{P}_{\text{r}}$ and $\boldsymbol{P}_{\text{t}}$ and recursively applying the equation at these points. The overall process is sometimes represented as a tree as we indicated in Figure 10.5.

A procedure to implement ray tracing is easily written and has low code complexity. The top-level procedure calls itself to calculate the reflected and transmitted rays. The geometric calculation for the reflected and transmitted ray directions are given in Chapter 1, and details of intersection testing a ray with a sphere will also be found there.



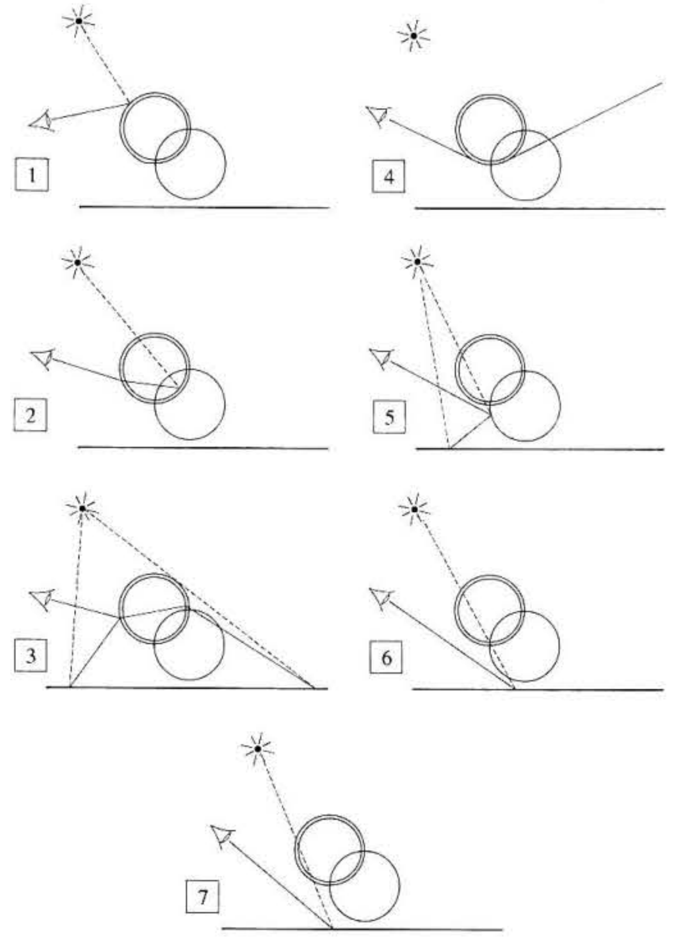


Figure 12.4
The Whitted scene (see also Colour Plate section).

The basic control procedure for a ray tracer consists of a simple recursive procedure that reflects the action at a node where, in general, two rays are spawned. Thus the procedure will contain two calls to itself, one for the transmitted and one for the reflected ray. We can summarize the action as:

```
ShootRay (ray structure)
    intersection test
    if ray intersects an object
        get normal at intersection point
        calculate local intensity ($I_{local}$)
        decrement current depth of trace
        if depth of trace > 0
            calculate and shoot the reflected ray
            calculate and shoot the refracted ray
```

where the last two lines imply a recursive call of ShootRay(). This is the basic control procedure. Around the recursive calls there has to be some more detail which is:

```
Calculate and shoot reflected ray elaborates as
    if object is a reflecting object
        calculate reflection vector and include in the ray structure
        Ray Origin := intersection point
        Attenuate the ray (multiply the current $k_{rg}$ by its value at the
        previous invocation)
        ShootRay(reflected ray structure)
        if reflected ray intersects an object
            combine colours ($k_{rg}$ $I$) with $I_{local}$
```

```
Calculate and shoot refracted ray elaborates as
    if object is a refracting object
        if ray is entering object
            accumulate refractive index
            increment number of objects that the ray is currently
            inside
            calculate refraction vector and include in refracted ray
            structure
        else
            de-accumulate refractive index
            decrement number of objects that the ray is currently
            inside
            calculate refraction vector and include in refracted ray
            structure
        Ray origin := intersection point
        Attenuate ray ($k_{tg}$)
        if refracted ray intersects an object
            combine colours ($k_{tg}$ $I$) with $I_{local}$
```

The ray structure needs to contain at least the following information:

- Origin of the ray.
- Its direction.
- Its intersection point.
- Its current colour at the intersection point.
- Its current attenuation.
- The distance of the intersection point from the ray origin.
- The refractive index the ray is currently experiencing.
- Current depth of the trace.
- Number of objects we are currently inside.

Thus the general structure is of a procedure calling itself twice for a reflected and refracted ray. The first part of the procedure finds the object closest to the ray start. Then we find the normal and apply the local shading model, attenuating the light source intensity if there are any objects between the intersection point $\boldsymbol{P}$ and the object. We then call the procedure recursively for the reflected and transmitted ray.

The number of recursive invocations of ShootRay() is controlled by the depth of trace parameter. If this is unity the scene is rendered just with a local reflection model. To discover any reflections of another object at a point $\boldsymbol{P}$ we need a depth of at least two. To deal with transparent objects we need a depth of at least three. (The initial ray, the ray that travels through the object and the emergent ray have to be followed. The emergent ray returns an intensity from any object that it hits.)

12.3 The adventures of seven rays – a ray tracing study

Return to Figure 12.4. We consider the way in which the ray tracing model works in the context of the seven pixels shown highlighted. The scene itself consists of a thin walled or hollow sphere, that is almost perfectly transparent, together with a partially transparent white sphere, both of which are floating above the ubiquitous red and yellow chequerboard. Everywhere else in object space is a blue background. The object properties are summarized in Table 12.1. Note that this model allows us to set k_s to a different value from k_{rg} – the source of the contradiction mentioned in Section 12.1.2; reflected rays are treated differently depending on which component (local or global) is being considered.

Consider the rays associated with the pixels shown in Figure 10.4.

Ray 1

This ray is along a direction where a specular highlight is seen on the highly transparent sphere. Because the ray is near the mirror direction of $\boldsymbol{L}$, the contribution from the specular component in $I_{\text{local}}(\boldsymbol{P})$ is high and the contributions

Table 12.1

Very transparent hollow sphere						
k_d (local)	0.1	0.1	0.1	(low)		
k_s (local)	0.8	0.8	0.8	(high)		
k_{rg}	0.1	0.1	0.1	(low)		
k_{tg}	0.9	0.9	0.9	(high)		
Opaque (white) sphere						
k_d (local)	0.2	0.2	0.2	(white)		
k_s (local)	0.8	0.8	0.8	(white)		
k_{rg}	0.4	0.4	0.4	(white)		
k_{tg}	0.0	0.0	0.0			
Chequerboard						
k_d (local)	1.0	0.0	0.0/1.0	1.0	0.0	(high red or yellow)
k_s (local)	0.2	0.2	0.2			
k_{rg}	0					
k_{tg}	0					
Blue background						
k_d (local)	0.1	0.1	1.0	(high blue)		
Ambient light	0.3	0.3	0.3			
Light	0.7	0.7	0.7			

from $k_{rg}I(\boldsymbol{P}_r)$ is low. For this object k_d, the local diffuse coefficient is low (it is multiplied by 1 – transparency value) and k_s is high with respect to k_{rg}. However, note that the local contribution only dominates over a very small area of the surface of the object. Also note that, as we have already mentioned, the highlight should not be spread. But if we left it as occupying a single pixel it would not be visible.

Ray 2

Almost the same as ray 1 except that the specular highlight appears on the inside wall of the hollow sphere. This particular ray demonstrates another accepted error in ray tracing. Effectively the ray from the light travels through the sphere without refracting (that is, we simply compare $\boldsymbol{L}$ with the local value of $\boldsymbol{N}$ and ignore the fact that we are now inside a sphere). This means that the specular highlight is in the *wrong* position but we simply accept this because we have no intuitive expectation of the correct position anyway. We simply accept it to be correct.

Ray 3

Ray 3 also hits the thin-walled sphere. The local contribution at all hits with the hollow sphere are zero and the predominant contribution is the chequerboard.

This is subject to slight distortion due to the refractive effect of the sphere walls. The red (or yellow) colour comes from the high k_d in $I_{local}(\boldsymbol{P})$ where $\boldsymbol{P}$ is a point on the chequerboard. k_{rg} and k_{tg} are zero for this surface. Note, however, that we have a mix of two chequerboards. One is as described and the other is the superimposed reflection on the outside surface of the sphere.

Ray 4
Again this hits the thin-walled sphere, but this time in a direction where the distance travelled through the glass is significant (that is, it only travels through the glass it does not hit the air inside) causing a high refractive effect and making the ray terminate in the blue background.

Ray 5
This ray hits the opaque sphere and returns a significant contribution from the local component due to a white k_d (local). At the first hit the global reflected ray hits the chequerboard. Thus there is a mixture of:

white (from the sphere's diffuse component)
red/yellow (reflected from the chequerboard)

Ray 6
This ray hits the chequerboard initially and the colour comes completely from the local component for that surface. However, the point is in shadow and this is discovered by the intersection of the ray $\boldsymbol{L}$ and the opaque sphere.

Ray 7
The situation with this ray is exactly the same as for ray 6 except that it is the thin walled sphere that intersects $\boldsymbol{L}$. Thus the shadow area intensity is not reduced by as much as the previous case. Again we do not consider the recursive effect that $\boldsymbol{L}$ would in fact experience and so the shadow is in effect in the wrong place.

12.4 Ray tracing polygon objects – interpolation of a normal at an intersection point in a polygon

Constraining a modelling primitive to be a sphere or at best a quadric solid is hopelessly restrictive in practice and in this section we will look at ray tracing polygonal objects. Extending the above program to cope with general polygon objects requires the development of an intersection test for polygons (see Section 1.4.3) and a method of calculating or interpolating a normal at the hit point $\boldsymbol{P}$. We remind ourselves that the polygonal facets are only approximations to a curved surface and, just as in Phong shading we need to interpolate, from the vertex normals, an approximation to the surface normal of the 'true' surface that

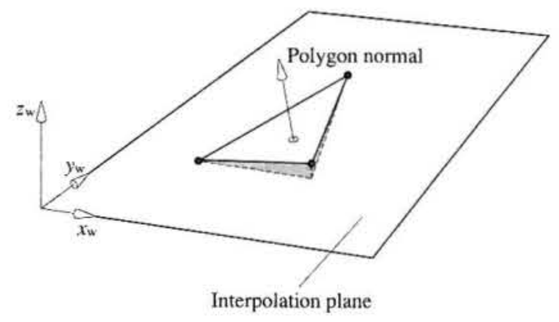


Figure 12.5
A polygon that lies almost in the $x_w y_w$ plane will have a high z_w component. We choose this plane in which to perform interpolation of vertex normals.

the facet approximates. This entity is required for the local illumination component and to calculate reflection and refraction. Recall that in Phong interpolation (see Section 6.3.2) we used the two-dimensional component of screen space to interpolate, pixel by pixel, scan line by scan line, the normal at each pixel projection on the polygon. We interpolated three of the vertex normals using two-dimensional screen space as the interpolation basis. How do we interpolate from the vertex normals in a ray tracing algorithm, bearing in mind that we are operating in world space? One easy approach is to store the polygon normal for each polygon as well as its vertex normals. We find the largest of its three components x_w, y_w and z_w. The largest component identifies which of the three world coordinate planes the polygon is closest to in orientation, and we can use this plane in which to interpolate using the same interpolation scheme as we employed for Phong interpolation (see Section 1.5). This plane is equivalent to the use of the screen plane in Phong interpolation. The idea is shown in Figure 12.5. This plane is used for the interpolation as follows. We consider the polygon to be represented in a coordinate system where the hit point $\boldsymbol{P}$ is the origin. We then have to search the polygon vertices to find the edges that cross the 'medium' axis. This enables us to interpolate the appropriate vertex normals to find $\boldsymbol{N}_a$ and $\boldsymbol{N}_b$ from which we find the required normal $\boldsymbol{N}_p$ (Figure 12.6). Having found the interpolated normal we can calculate the local illumination component and the reflected and the refracted rays. Note that because we

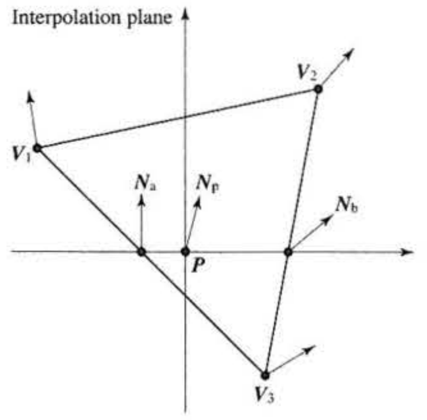


Figure 12.6
Finding an interpolated normal at a hit point $\boldsymbol{P}$.

are 'randomly' interpolating we lose the efficiency advantages of the Phong interpolation, which was incremental on a pixel by pixel, scan line by scan line basis.

We conclude that in ray tracing polygonal objects we incur two significant costs. First, the more overwhelming cost is that of intersection testing each polygon in an object. Second, we have the cost of finding an interpolated normal on which to base our calculations.

12.5 Efficiency measures in ray tracing

12.5.1 Adaptive depth control

The trace depth required in a ray tracing program depends upon the nature of the scene. A scene containing highly reflective surfaces and transparent objects will require a higher maximum depth than a scene that consists entirely of poorly reflecting surfaces and opaque objects. (Note that if the depth is set equal to unity then the ray tracer functions exactly as a conventional renderer, which removes hidden surfaces and applies a local reflection model.)

It is pointed out in Hall and Greenberg (1983) that the percentage of a scene that consists of highly transparent and reflective surfaces is, in general, small and it is thus inefficient to trace every ray to a maximum depth. Hall and Greenberg suggest using an adaptive depth control that depends on the properties of the materials with which the rays are interacting. The context of the ray being traced now determines the termination depth, which can be any value between unity and the maximum pre-set depth.

Rays are attenuated in various ways as they pass through a scene. When a ray is reflected at a surface, it is attenuated by the global specular reflection coefficient for the surface. When it is refracted at a surface, it is attenuated by the global transmission coefficient for the surface. For the moment, we consider only this attenuation at surface intersections. A ray that is being examined as a result of backward tracing through several intersections will make a contribution to the top level ray that is attenuated by several of these coefficients. Any contribution from a ray at depth n to the colour at the top level is attenuated by the product of the global coefficients encountered at each node:

$$k_1k_2 \ldots k_{n-1}$$

If this value is below some threshold, there will be no point in tracing further.

In general, of course, there will be three colour contributions (RGB) for each ray and three components to each of the attenuation coefficients. Thus when the recursive procedure is activated it is given a cumulative weight parameter that indicates the final weight that will be given at the top level to the colour returned for the ray represented by that procedure activation. The correct weight for a new procedure activation is easily calculated by taking the cumulative weight for the ray currently being traced and multiplying it by the reflection or

transmission coefficient for the surface intersection at which the new ray is being created.

Another way in which a ray can be attenuated is by passing for some distance through an opaque material. This can be dealt with by associating a transmittance coefficient with the material composing an object. Colour values would then be attenuated by an amount determined by this coefficient and the distance a ray travels through the material. A simple addition to the intersection calculation in the ray tracing procedure would allow this feature to be incorporated.

The use of adaptive depth control will prevent, for example, a ray that initially hits an almost opaque object spawning a transmitted ray that is then traced through the object and into the scene. The intensity returned from the scene may then be so attenuated by the initial object that this computation is obviated. Thus, depending on the value to which the threshold is pre-set, the ray will, in this case, be terminated at the first hit.

For a highly reflective scene with a maximum tree depth of 15, Hall and Greenberg report (1983) that this method results in an average depth of 1.71, giving a large potential saving in image generation time. The actual saving achieved will depend on the nature and distribution of the objects in the scene.

12.5.2 First hit speed up

In the previous section it was pointed out that even for highly reflective scenes, the average depth to which rays were traced was between one and two. This fact led Weghorst *et al.* (1984) to suggest a hybrid ray tracer, where the intersection of the initial ray is evaluated during a preprocessing phase, using a hidden surface algorithm. The implication here is that the hidden surface algorithm will be more efficient than the general ray tracer for the first hit. Weghorst *et al.* (1984) suggest executing a modified Z-buffer algorithm, using the same viewing parameters. Simple modifications to the Z-buffer algorithm will make it produce, for each pixel in the image plane, a pointer to the object visible at that pixel. Ray tracing, incorporating adaptive depth control then proceeds from that point. Thus the expensive intersection tests associated with the first hit are eliminated.

12.5.3 Bounding objects with simple shapes

Given that the high cost of ray tracing is embedded in intersection testing, we can greatly increase the efficiency of a recursive ray tracer by making this part of the algorithm as efficient as possible. An obvious and much used approach is to enclose the object in a 'simple' volume known as a bounding volume. Initially we test the ray for intersection with a bounding volume and only if the ray enters this volume do we test for intersection with the object. Note that we also used this approach in the operation of culling against a view volume (see Chapter 6) and in collision detection (see Chapter 17).

Two properties are required of a bounding volume. First, it should have a simple intersection test – thus a sphere is an obvious candidate. Second, it should efficiently enclose the object. In this aspect a sphere is deficient. If the object is long and thin the sphere will contain a large void volume and many rays will pass the bounding volume test but will not intersect the object. A rectangular solid, where the relative dimensions are adjustable, is possibly the best simple bounding volume. (Details of intersection testing of both spheres and boxes are given in Chapter 1.)

The dilemma of bounding volumes is that you cannot allow the complexity of the bounding volume scheme to grow too much, or it obviates its own purpose. Usually for any scene, the cost of bounding volume calculations will be related to their enclosing efficiency. This is easily shown conceptually. Figure 12.7 shows a two-dimensional scene containing two rods and a circle representing complex polygonal objects. Figure 12.7(a) shows circles (spheres) as bounding volumes with their low enclosing efficiency for the rods. Not only are the spheres inefficient, but they intersect each other, and the space occupied by other objects. Using boxes aligned with the scene axes (axis aligned bounding boxes, or AABBs) is better (Figure 12.7(b)) but now the volume enclosing the sloping rod is inefficient. For this scene the best bounding volumes are boxes with any orientation (Figure 12.7(c)); the cost of testing the bounding volumes increases from spheres to boxes with any orientation. These are known as OBBs.

Weghorst *et al.* (1984) define a 'void' area, of a bounding volume, to be the difference in area between the orthogonal projections of the object and bounding volume onto a plane perpendicular to the ray and passing through the origin of the ray (see Figure 12.8). They show that the void area is a function of object, bounding volume and ray direction and define a cost function for an intersection test:

$$T = b^*B + i^*I$$

where:

T is the total cost function
b is the number of times that the bounding volume is tested for intersection
B is the cost of testing the bounding volume for intersection
i is the number of times that the item is tested for intersection (where $i \le b$)
I is the cost of testing the item for intersection

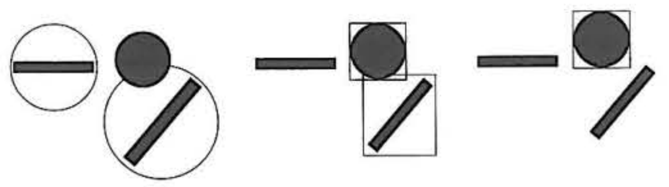


Figure 12.7
Three different bounding volumes, going from (a) to (c). The complexity cost of the bounding volume increases together with its enclosing efficiency.
(a) Circles (spheres) as bounding volumes;
(b) rectangles (boxes) as bounding volumes;
(c) rectangles (boxes) at any orientation.

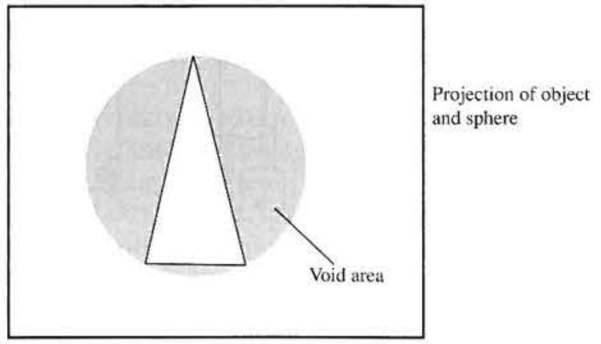


Figure 12.8
The void area of a bounding sphere.

It is pointed out by the authors that the two products are generally interdependent. For example, reducing B by reducing the complexity of the bounding volume will almost certainly increase i. A quantitive approach to selecting the optimum of a sphere, a rectangular parallelepiped and a cylinder as bounding volumes is given.

12.5.4 Secondary data structures

Another common approach to efficiency in intersection testing is to set up a secondary data structure to control the intersection testing. The secondary data structure is used as a guide and the primary data structure – the object database – is entered at the most appropriate point.

Bounding volume hierarchies

A common extension to bounding volumes, first suggested by Rubin and Whitted (1980) and discussed in Weghorst *et al.* (1984), is to attempt to impose a hierarchical structure of such volumes on the scene. If it is possible, objects in close spatial proximity are allowed to form clusters, and the clusters are themselves enclosed in bounding volumes. For example, Figure 12.9 shows a container (a) with one large object (b) and four small objects (c_1, c_2, c_3 and c_4) inside it. The tree represents the hierarchical relationship between seven boundary extents: a cylinder enclosing all the objects, a cylinder enclosing (b), a cylinder enclosing (c_1, c_2, c_3, c_4) and the bounding cylinders for each of these objects. A ray traced against bounding volumes means that such a tree is traversed from the topmost level. A ray that happened to intersect c_1 in the above example would, of course, be tested against the bounding volumes for c_1, c_2, c_3 and c_4, but only because it intersects the bounding volume representing that cluster. This example also demonstrates that the nature of the scene should enable reasonable clusters of adjacent objects to be selected, if substantial savings over a non-hierarchical bounding scheme are to be achieved. Now the intersection test is implemented as a recursive process, descending through a hierarchy, only from

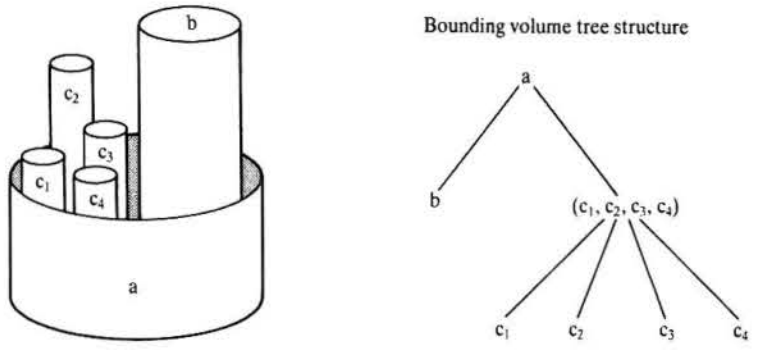


Figure 12.9
A simple scene and the associated bounding cylinder tree structure.

those nodes where intersections occur. Thus a scene is grouped, where possible, into object clusters and each of those clusters may contain other groups of objects that are spatially clustered. Ideally, high-level clusters are enclosed in bounding volumes that contain lower-level clusters and bounding volumes. Clusters can only be created if objects are sufficiently close to each other. Creating clusters of widely separated objects obviates the process. The potential clustering and the depth of the hierarchy will depend on the nature of the scene: the deeper the hierarchy the more the potential savings. The disadvantage of this approach is that it depends critically on the nature of the scene. Also, considerable user investment is required to set up a suitable hierarchy.

Bounding volume hierarchies used in collision detection are discussed in Chapter 17. Although identical in principle, collision detection requires efficient testing for intersection between pairs of bounding volumes, rather than ray/volume testing. OBB hierarchies have proved useful in this and are described in Section 17.5.2.

The use of spatial coherence

Currently, spatial coherence is the only approach that looks like making ray tracing a practical proposition for routine image synthesis. For this reason it is discussed in some detail. Object coherence in ray tracing has generally been ignored. The reason is obvious. By its nature a ray tracing algorithm spawns rays of arbitrary direction anywhere in the scene. It is difficult to use such 'random' rays to access the object data structure and efficiently extract those objects in the path of a ray. Unlike an image space scan conversion algorithm where, for example, active polygons can be listed, there is no *a priori* information on the sequence of rays that will be spawned by an initial or view ray. Naive ray tracing algorithms execute an exhaustive search of all objects after each hit, perhaps modified by a scheme such as bounding volumes, to constrain the search.

The idea behind spatial coherence schemes is simple. The space occupied by the scene is subdivided into regions. Now, rather than check a ray against all objects or sets of bounded objects, we attempt to answer the question: is the

region, through which the ray is currently travelling, occupied by any objects? Either there is nothing in this region, or the region contains a small subset of the objects. This group of objects is then tested for intersection with the ray. The size of the subset and the accuracy to which the spatial occupancy of the objects is determined varies, depending on the nature and number of the objects and the method used for subdividing the space.

This approach, variously termed spatial coherence, spatial subdivision or space tracing has been independently developed by several workers, notably Glassner (1984), Kaplan (1985) and Fujimoto *et al.* (1986). All of these approaches involve pre-processing the space to set up an auxiliary data structure that contains information about the object occupancy of the space. Rays are then traced using this auxiliary data structure to enter the object data structure. Note that this philosophy (of pre-processing the object environment to reduce the computational work required to compute a view) was first employed by Schumaker *et al.* (1969) in a hidden surface removal algorithm developed for flight simulators (see Section 6.6.10). In this algorithm, objects in the scene are clustered into groups by subdividing the space with planes. The spatial subdivision is represented by a binary tree. Any view point is located in a region represented by a leaf in the tree. An on-line tree traversal for a particular view point quickly yields a depth priority order for the group clusters. The important point about this algorithm is that the spatial subdivision is computed off-line and an auxiliary structure, the binary tree representing the subdivision, is used to determine an initial priority ordering for the object clusters. The motivation for this work was to speed up the on-line hidden surface removal processing and enable image generation to work in real time.

Dissatisfaction with the bounding volume or extent approach, to reducing the number of ray object intersection tests, appears in part to have motivated the development of spatial coherence methods (Kaplan 1985). One of the major objections to bounding volumes has already been pointed out. Their 'efficiency' is dependent on how well the object fills the space of the bounding volume. A more fundamental objection is that such a scheme may increase the efficiency of the ray–object intersection search, but it does nothing to reduce the dependence on the number of objects in the scene. Each ray must still be tested against the bounding extent of every object and the search time becomes a function of scene complexity. Also, although major savings can be achieved by using a hierarchical structure of bounding volumes, considerable investment is required to set up an appropriate hierarchy, and depending on the nature and disposition of objects in the scene, a hierarchical description may be difficult or impossible. The major innovation of methods described in this section is to make the rendering time constant (for a particular image space resolution) and eliminate its dependence on scene complexity.

The various schemes that use the spatial coherence approach differ mainly in the type of auxiliary data structure used. Kaplan (1985) lists six properties that a practical ray tracing algorithm should exhibit if the technique is to be used in routine rendering applications. Kaplan's requirements are:

(1) Computation time should be relatively independent of scene complexity (number of objects in the environment, or complexity of individual objects), so that scenes having realistic levels of complexity can be rendered.
(2) Per ray time should be relatively constant, and not dependent on the origin or direction of the ray. This property guarantees that overall computation time for a shaded image will be dependent only on overall image resolution (number of first-level rays traced) and shading effects (number of second-level and higher level rays traced). This guarantees predictable performance for a given image resolution and level of realism.
(3) Computation time should be 'rational' and 'interactive' (within a few minutes) on affordable processor systems.
(4) The algorithm should not require the user to supply hierarchical object descriptions or object clustering information. The user should be able to combine data generated at different times, and by different means, into a single scene.
(5) The algorithm should deal with a wide variety of primitive geometric types, and should be easily extensible to new types.
(6) The algorithm's use of coherence should not reduce its applicability to parallel processing or other advanced architectures. Instead, it should be amenable to implementation on such architectures.

Kaplan summarizes these requirements by saying, 'in order to be really usable, it must be possible to trace a large number of rays in a complex environment in a rational, predictable time, for a reasonable cost'.

Two related approaches to an auxiliary data structure have emerged. These involve an octree representation (Fujimoto *et al.* 1986; Glassner 1984) and a data structure called a BSP (binary space partitioning). The BSP tree was originally proposed by Fuchs (1980) and is used in Kaplan (1985).

Use of an octree in ray tracing

An octree (see Chapter 2) is a representation of the objects in a scene that allows us to exploit spatial coherence – objects that are close to each other in space are represented by nodes that are close to each other in the octree.

When tracing a ray, instead of doing intersection calculations between the ray and every object in the scene, we can now trace the ray from subregion to subregion in the subdivision of occupied space. For each subregion that the ray passes through, there will only be a small number of objects (typically one or two) with which it could intersect. Provided that we can rapidly find the node in the octree that corresponds to a subregion that a ray is passing through, we have immediate access to the objects that are on, or close to, the path of the ray. Intersection calculations need only be done for these objects. If space has been subdivided to a level where each subregion contains only one or two objects, then the number of intersection tests required for a region is small and does not tend to increase with the complexity of the scene.

Tracking a ray using an octree

In order to use the space subdivision to determine which objects are close to a ray, we must determine which subregion of space the ray passes through. This involves tracking the ray into and out of each subregion in its path. The main operation required during this process is that of finding the node in the octree, and hence the region in space, that corresponds to a point (x, y, z).

The overall tracking process starts by detecting the region that corresponds to the start point of the ray. The ray is tested for intersection with any objects that lie in this region and if there are any intersections, then the first one encountered is the one required for the ray. If there are no intersections in the initial region, then the ray must be tracked into the next region through which it passes. This is done by calculating the intersection of the ray with the boundaries of the region and thus calculating the point at which the ray leaves the region. A point on the ray a short distance into the next region is then used to find the node in the octree that corresponds to the next region. Any objects in this region are then tested for intersections with the ray. The process is repeated as the ray tracks from region to region until an intersection with an object is found or until the ray leaves occupied space.

The simplest approach to finding the node in the octree that corresponds to a point (x, y, z) is to use a data structure representation of the octree to guide the search for the node. Starting at the top of the tree, a simple comparison of coordinates will determine which child node represents the subregion that contains the point (x, y, z). The subregion, corresponding to the child node, may itself have been subdivided and another coordinate comparison will determine which of its children represents the smaller subregion that contains (x, y, z). The search proceeds down the tree until a terminal node is reached. The maximum number of nodes traversed during this search will be equal to the maximum depth of the tree. Even for a fairly fine subdivision of occupied space, the search length will be short. For example, if the space is subdivided at a resolution of $1024 \times 1024 \times 1024$, then the octree will have depth 10 ($= \log_8(1024 \times 1024 \times 1024)$).

So far we have described a simple approach to the use of an octree representation of space occupancy to speed up the process of tracking a ray. Two variations of this basic approach are described by Glassner (1984) and Fujimoto *et al.* (1986). Glassner describes an alternative method for finding the node in the octree corresponding to a point (x, y, z). In fact, he does not store the structure of the octree explicitly, but accesses information about the voxels via a hash table that contains an entry for each voxel. The hash table is accessed using a code number calculated from the (x, y, z) coordinates of a point. The overall ray tracking process proceeds as described in our basic method.

In Fujimoto *et al.* (1986) another approach to tracking the ray through the voxels in the octree is described. This method eliminates floating point multiplications and divisions. To understand the method it is convenient to start by ignoring the octree representation. We first describe a simple data structure representation of a space subdivision called SEADS (Spatially Enumerated Auxiliary Data Structure). This involves dividing all of occupied space into equally sized voxels regardless of occupancy by objects. The three-dimensional grid obtained in this way is analogous to that obtained by the subdivision of a two-dimensional graphics screen into pixels. Because regions are subdivided regardless of occupancy by objects, a SEADS subdivision generates many more voxels than the octree subdivision described earlier. It thus involves 'unnecessary' demands for storage space. However, the use of a SEADS enables very fast tracking of rays from region to region. The tracking algorithm used is an extension of the DDA (Digital Differential Analyzer) algorithm used in two-dimensional graphics for selecting the sequence of pixels that represent a straight line between two given end points. The DDA algorithm used in two-dimensional graphics selects a subset of the pixels passed through by a line, but the algorithm can easily be modified to find all the pixels touching the line. Fujimoto *et al.* (1986) describe how this algorithm can be extended into three-dimensional space and used to track a ray through a SEADS three-dimensional grid. The advantage of the '3D-DDA' is that it does not involve floating point multiplication and division. The only operations involved are addition, subtraction and comparison, the main operation being integer addition on voxel coordinates.

The heavy space overheads of the complete SEADS structure can be avoided by returning to an octree representation of the space subdivision. The 3D-DDA algorithm can be modified so that a ray is tracked through the voxels by traversing the octree. In the octree, a set of eight nodes with a common parent node represents a block of eight adjacent cubic regions forming a 2 × 2 × 2 grid. When a ray is tracked from one region to another within this set, the 3D-DDA algorithm can be used without alteration. If a ray enters a region that is not represented by a terminal node in the tree, but is further subdivided, then the subregion that is entered is found by moving down the tree. The child node required at each level of descent can be discovered by adjusting the control variables of the DDA from the level above. If the 3D-DDA algorithm tracks a ray out of the 2 × 2 × 2 region currently being traversed, then the octree must be traversed upwards to the parent node representing the complete region. The 3D-DDA algorithm then continues at this level, tracking the ray within the set of eight regions containing the parent region. The upward and downward traversals of the tree involve multiplication and division of the DDA control variables by 2, but this is a cheap operation.

Finally, we summarize and compare the three spatial coherence methods by listing their most important efficiency attributes:

- Octrees: are good for scenes whose occupancy density varies widely – regions of low density will be sparsely subdivided, high density regions will be finely subdivided. However, it is possible to have small objects in large regions. Stepping from region to region is slower than with the other two methods because the trees tend to be unbalanced.

- SEADS: stepping is faster than an octree but massive memory costs are incurred by the secondary data structure.
- BSP: the depth of the tree is smaller than an octree for most scenes because the tree is balanced. Octree branches can be short, or very long for regions of high spatial occupancy. The memory costs are generally lower than those of an octree. Void areas will tend to be smaller.

12.5.5 Ray space subdivision

In this unique scheme, suggested by Arvo and Kirk (1987), instead of subdividing object space according to occupancy, ray space is subdivided into five-dimensional hypercubic regions. Each hypercube in five-dimensional space is associated with a candidate list of objects for intersection. That stage in object space subdivision schemes where three-space calculations have to be invoked to track a ray through object space is now eliminated. The hypercube that contains the ray is found and this yields a complete list of all the objects that can intersect the ray. The cost of the intersection testing is now traded against higher scene pre-processing complexity.

A ray can be considered as a single point in five-dimensional space. It is a line with a three-dimensional origin together with a direction that can be specified by two angles in a unit sphere. Instead of using a sphere to categorize direction, Arvo and Kirk (1987) use a 'direction cube'. (This is exactly the same tool as the light buffer used by Haines and Greenberg (1986) – see Section 12.1.3.) A ray is thus specified by the 5-tuple (x, y, z, u, v), where x, y, z is the origin of the ray and u, v the direction coordinates; together with a cube face label that indicates which face of the direction cube the ray passes through. Six copies of a five-dimensional hypercube (one for each direction cube face) thus specify a collection of rays having similar origins and similar directions.

This space is subdivided according to object occupancy and candidate lists are constructed for the subdivided regions. A 'hyper-octree' – a five-dimensional analog of an octree – is used for the subdivision.

To construct candidate lists as five-dimensional space is subdivided, the three-dimensional equivalent of the hypercube must be used in three-space. This is a 'beam' or an unbounded three-dimensional volume that can be considered the union of the volume of ray origins and the direction pyramid formed by a ray origin and its associated direction cell (Figure 12.10). Note that the beams in three-space will everywhere intersect each other, whereas their hypercube equivalents in five-space do not intersect. This is the crux of the method – the five-space can be subdivided and that subdivision can be acheived using binary partitioning. However, the construction of the candidate lists is now more difficult than with object space subdivision schemes. The beams must be intersected with the bounding volumes of objects. Arvo and Kirk (1987) report that detecting polyhedral intersections is too costly and suggest the approximation where beams are represented or bounded by cones interacting with spheres as object bounding volumes.

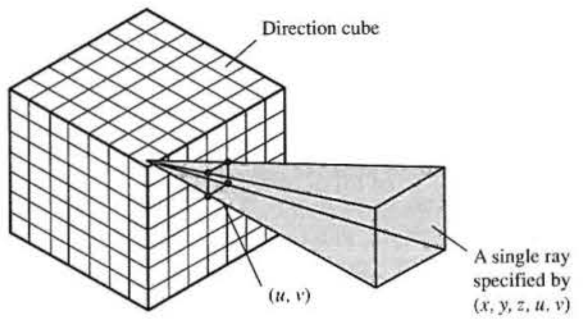


Figure 12.10
A ray (or beam) as a single point in (*x*, *y*, *z*, *u*, *v*) space.

12.6 The use of ray coherence

Up to now we have considered a ray to be infinitesimally thin and looked at efficiency measures that attempt to speed up the basic algorithm. It is easy to see that a major source of inefficiency that we have not touched on until now is the lack of use of ray coherence. This simply means that if the ray tracing algorithm generates a ray for each pixel and separately traces every such ray we are taking no account whatever of the fact that adjacent initial rays will tend to follow the same path. We will now look at ways in which we can 'broaden' a ray into a geometric entity.

Heckbert and Hanrahan (1984) exploit the coherence that is available from the observation that, for any scene, a particular ray has many neighbours each of which tends to follow the same path. Rather than tracing single rays, then, why not trace groups of parallel rays, sharing the intersection calculations over a bundle of rays? This is accomplished by recursively applying a version of the Weiler–Atherton hidden surface removal algorithm (Weiler and Atherton 1977). The Weiler–Atherton algorithm is a projection space subdivision algorithm involving a preliminary depth sort of polygons followed by a sort of the fragments generated by clipping the sorted polygons against each other. Finally, recursive subdivision is used to sort out any remaining ambiguities. This approach restricts the objects to be polygonal, thus destroying one of the important advantages of a ray tracer which is that different object definitions are easily incorporated due to the separation of the intersection test from the ray tracer.

The initial beam is the viewing frustum. This beam or bundle of rays is traced through the environment and is used to build an intersection tree, different from a single ray tree in that a beam may intersect many surfaces rather than one. Each node in the tree now contains a list of surfaces intersected by the beam.

The procedure is carried out in a transformed coordinate system called the beam coordinate system. Initially this is the view or eye coordinate system. Beams are volumes swept out as a two-dimensional polygon in the *xy* plane is translated along the *z* axis.

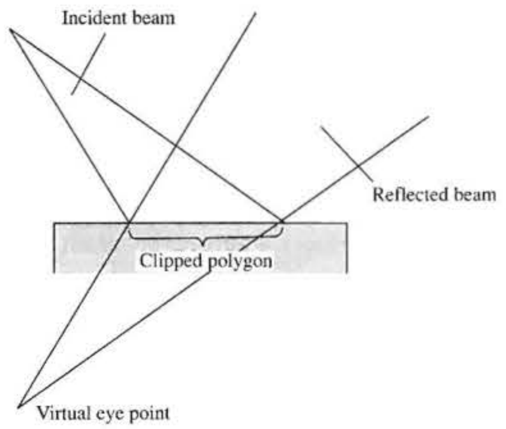


Figure 12.11
Reflection in beam tracing.

Reflection (and refraction) are modelled by calling the beam tracer recursively. A new beam is generated for each beam–object intersection. The cross-section of any refelected beam is defined by the area of the polygon clipped by the incident beam and a virtual eye point (Figure 12.11).

Apart from the restriction to polygonal objects the approach has other disadvantages. Beams that partially intersect objects change into beams with complex cross-sections. A cross-section can become disconnected or may contain a hole (Figure 12.12). Another disadvantage is that refraction is a non-linear phenomenon and the geometry of a refracted beam will not be preserved. Refraction therefore, has to be approximated using a linear transformation.

Another approach to beam tracing is the pencil technique of Shinya *et al.* (1987). In this method a pencil is formed from rays called 'paraxial rays'. These are rays that are near to a reference ray called an axial ray. A paraxial ray is represented by a four-dimensional vector in a coordinate system associated with the axial ray. Paraxial approximation theory, well known in optical design and electromagnetic analysis, is then used to trace the paraxial rays through the environment. This means that for any rays that are near the axial ray, the pencil transformations are linear and are 4×4 matrices. Error analysis in paraxial theory supplies functions that estimate errors and provide a constraint for the spread angle of the pencil.

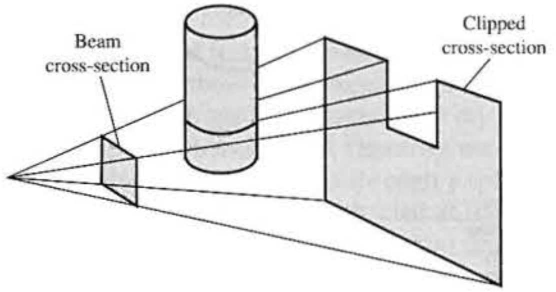


Figure 12.12
A beam that partially intersects an object produces a fragmented cross-section.

The 4 × 4 system matrices are determined by tracing the axial ray. All the paraxial rays in the pencil can then be traced using these matrices. The paraxial approximation theory depends on surfaces being smooth so that a paraxial ray does not suddenly diverge because a surface discontinuity has been encountered. This is the main disadvantage of the method.

An approach to ray coherence that exploits the similarity between the intersection trees generated by successive rays is suggested by Speer *et al.* (1986). This is a direct approach to beam tracing and its advantage is that it exploits ray coherence without introducing a new geometrical entity to replace the ray. The idea here is to try to use the path (or intersection tree) generated by the previous ray, to construct the tree for the current ray (Figure 12.13). As the construction of the current tree proceeds, information from the corresponding branch of the previous tree can be used to predict the next object hit by the current ray. This means that any 'new' intervening object must be detected as shown in Figure 12.14. To deal with this, cylindrical safety zones are constructed around each ray in a ray set. A safety zone for ray$_{r-2}$ is shown in Figure 12.15. Now if the current ray does not pierce the cylinder of the corresponding previous ray, and this ray intersects the same object, then it cannot intersect any new intervening objects. If a ray does not pierce a cylinder, then new intersection tests are required as in standard ray tracing, and a new tree that is different from the previous tree, is constructed.

In fact, Speer *et al.* (1986) report that this method suffers from the usual computational cost paradox – the increase in complexity necessary to exploit the ray coherence properties costs more than the standard ray tracing as a function of scene complexity. This is despite the fact that two-thirds of the rays behave coherently. The reasons given for this are the cost of maintaining and pierce-checking the safety cylinders, whose average radius and length decrease as a function of scene complexity.

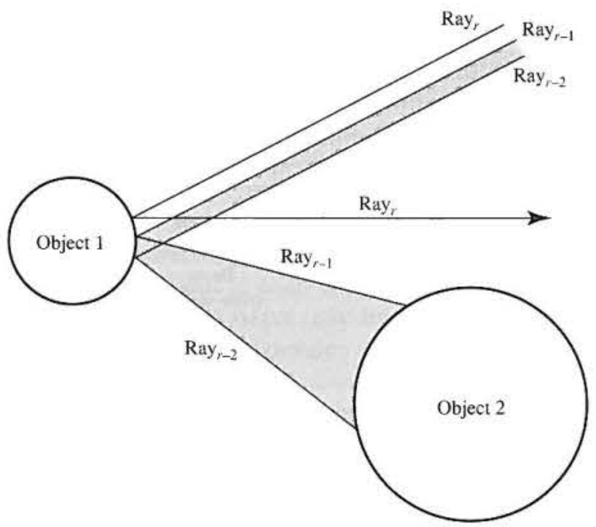
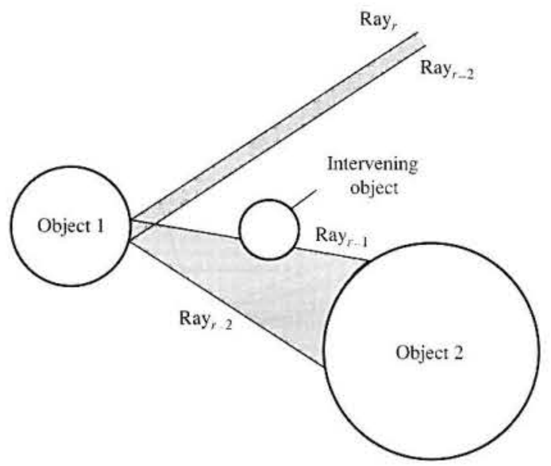


Figure 12.13
Ray coherence: the path of the previous ray can be used to predict the intersections of the current ray.

Figure 12.14
'Intervening' object in the path of the ray *r*.



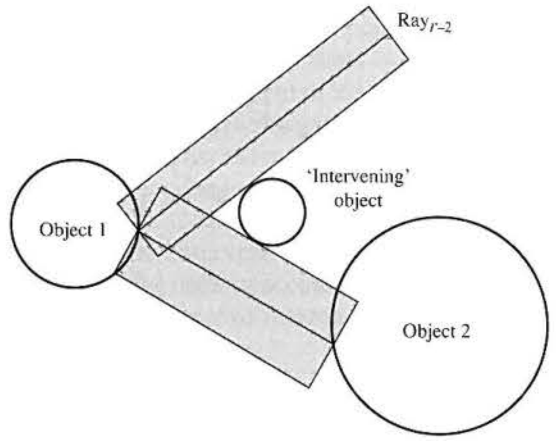


Figure 12.15
Cylindrical safety zones.

12.7 A historical digression – the optics of the rainbow

Many people associate the term 'ray tracing' with a novel technique but, in fact, it has always been part of geometric optics. For example, an early use of ray tracing in geometric optics is found in René Descartes' treatise, published in 1637, explaining the shape of the rainbow. From experimental observations involving a spherical glass flask filled with water, Descartes used ray tracing as a theoretical framework to explain the phenomenon. Descartes used the already known laws of reflection and refraction to trace rays through a spherical drop of water.

Rays entering a spherical water drop are refracted at the first air–water interface, internally reflected at the water–air interface and finally refracted as they emerge from the drop. As shown in Figure 12.16, horizontal rays entering the

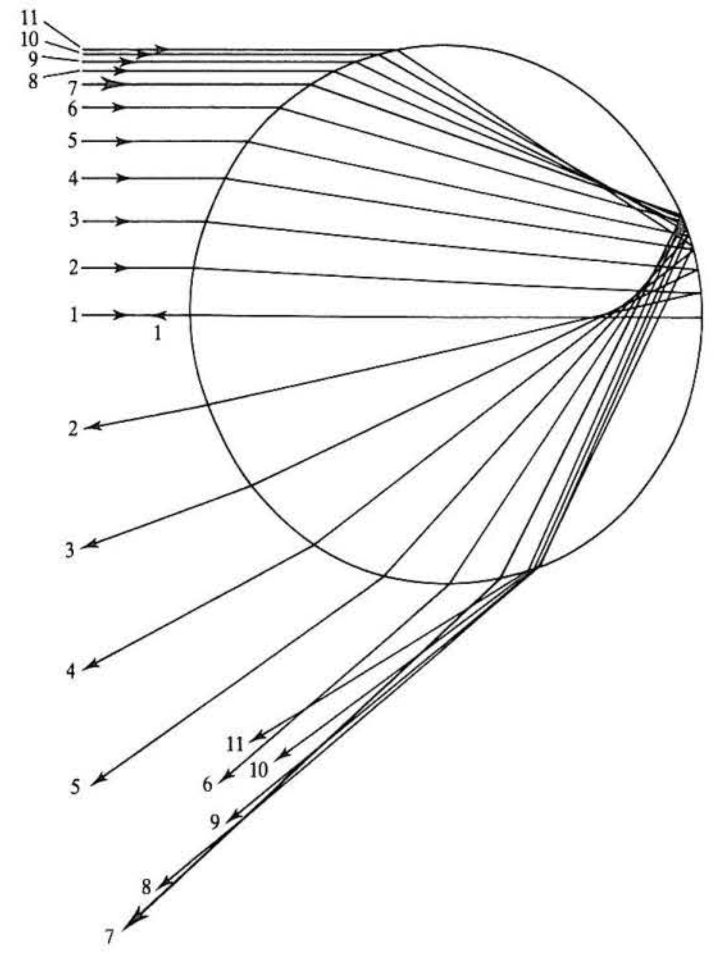


Figure 12.16
Tracing rays through a spherical water drop (ray 7 is the Descartes ray).

drop above the horizontal diameter emerge at an increasing angle with respect to the incident ray. Up to a certain maximum the angle of the exit ray is a function of the height of the incident ray above the horizontal diameter. This trend continues up to a certain ray, when the behaviour reverses and the angle between the incident and exit ray decreases. This ray is known as the Descartes ray, and at this point the angle between the incident and exit ray is 42°. Incident rays close to the Descartes ray emerge close to it and Figure 12.16 shows a concentration of rays around the exiting Descartes ray. It is this concentration of rays that makes the rainbow visible.

Figure 12.17 demonstrates the formation of the rainbow. An observer looking away from the sun sees a rainbow formed by '42°' rays from the sun. The paths of such rays form a 42° 'hemicone' centred at the observer's eye. (An interesting consequence of this model is that each observer has his own personal rainbow.)

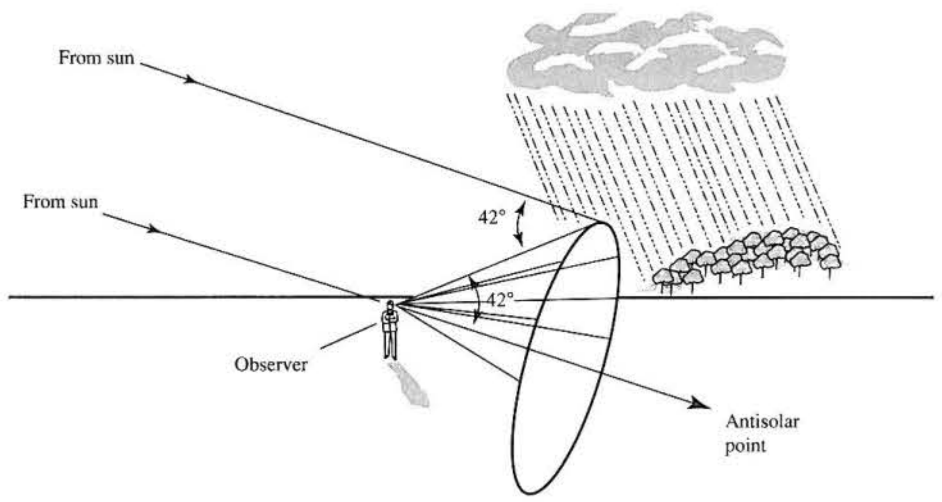


Figure 12.17
Formation of a rainbow.

This early, elegant use of ray tracing did not, however, explain that magical attribute of the rainbow – colour. Thirty years would elapse before Newton discovered that white light contained light at all wavelengths. Along with the fact that the refractive index of any material varies for light of different wavelengths, Descartes' original model is easily extended. About 42° is the maximum angle for red light, while violet rays emerge after being reflected and refracted through 40°. The model can then be seen as a set of concentric hemicones, one for each wavelength, centred on the observer's eye.

This simple model is also used to account for the fainter secondary rainbow. This occurs at 51° and is due to two internal reflections inside the water drops.

display will decrease the blue component, leaving the red and green components unchanged.

Gamma correction leaves zero and maximum intensities unchanged and alters the intensity in mid-range. A 'wrong' gamma that occurs either because gamma correction has not been applied or because an inaccurate value of gamma has been used in the correction will always result in a wrong image with respect to the calculated colour.

16 Image-based rendering and photo-modelling

Introduction

A new field with many diverse approaches, image-based rendering (IBR) is difficult to categorize. The motivation for the name is that most of the techniques are based on two-dimensional imagery, but this is not always the case and the way in which the imagery is used varies widely amongst methods. A more accurate common thread that runs through all the methods is pre-calculation. All methods make cost gains by pre-calculating a representation of the scene from which images are derived at run-time. IBR has mostly been studied for the common case of static scenes and a moving view point, but applications for dynamic scenes have been developed.

There is, however, no debate concerning the goal of IBR which is to decouple rendering time from scene complexity so that the quality of imagery, for a given frame time constraint, in applications like computer games and virtual reality can be improved over conventionally rendered scenes where all the geometry is reinserted into the graphics pipeline whenever a change is made to the view point. It has emerged, simultaneously with LOD approaches (see Chapter 2) and scene management techniques, as an effective means of tackling the dependency of rendering time on scene complexity.

We will also deal with photo-modelling in this chapter. This is related to image-based rendering because many image-based rendering schemes were designed to operate with photo-modelling. The idea of photo-modelling is to capture the real-world complexity and at the same time retain the flexibility advantages of three-dimensional graphics.

16.1 Reuse of previously rendered imagery – two-dimensional techniques

We begin by considering methods that rely on the concept of frame coherence and reuse of already rendered imagery in some way. Also, as the title of the section implies, we are going to consider techniques that are essentially two-dimensional. Although the general topic of image-based rendering, of course, itself implies two-dimensional techniques there has be some use of the depth information associated with the image, as we shall see in future sections. The distinction is that with techniques which we categorize as two-dimensional we do not operate with detailed depth values, for example, a value per pixel. We may only have a single depth value associated with the image entity as is implied by visibility ordering in image layers (see Section 16.2.2).

A useful model of an image-based renderer is to consider a required image being generated from a source or reference image – rendered in the normal way – by warping the reference image in image space (Figure 16.1). In this section we shall consider simple techniques based on texture mapping that can exploit the hardware facilities available on current 3D graphics cards. The novel approach here is that we consider rendered objects in the scene as texture maps, consider a texture map as a three-dimensional entity and pass it through the graphics pipeline. The common application of such techniques is in systems where a viewer moves through a static environment.

To a greater or lesser extent all such techniques involve some approximation compared with the projections that are computed using conventional techniques and an important part of such methods is determining when it is valid to reuse previously generated imagery and when new images must be generated.

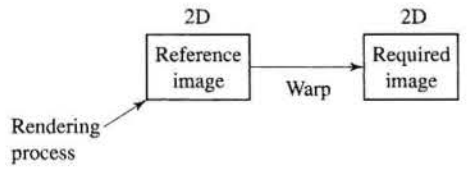


Figure 16.1
Planar imposters and image warping.

IBR as a process that produces an image by warping a reference image

16.1.1 Planar impostors or sprites

Impostor is the name usually given to an image of an object that is used in the form of a texture map – an entity we called a billboard in Chapter 8. In Chapter 8 the billboard was an object in its own right – it was a two-dimensional entity inserted into the scene. Impostors are generalizations of this idea. The idea is that because of the inherent coherence in consecutive frames in a moving view point sequence, the same impostor can be reused over a number of frames until an error measure exceeds some threshold. Such impostors are sometimes qualified by the adjective dynamic to distinguish them from pre-calculated object images that are not updated. A planar sprite is used as a texture map in a normal rendering engine. We use the adjective planar to indicate that no depth information is associated with the sprite – just as there is no depth associated with a texture map (although we retain depth information at the vertices of the rectangle that contains the sprite). The normal (perspective) texture mapping in the renderer takes care of warping the sprite as the view point changes.

There are many different possible ways in which sprites can be incorporated into a rendering sequence. Schaufler's method (Schaufler and Sturzlinger 1996) is typical and for generating an impostor from an object model it proceeds as follows. The object is enclosed in a bounding box which is projected onto the image plane resulting in the determination of the object's rectangular extent in screen space – for that particular view. The plane of the impostor is chosen to be that which is normal to the view plane normal and passes through the centre of the bounding box. The rectangular extent in screen space is initialized to transparent and the object rendered into it. This is then treated as a texture map and placed in the texture memory. When the scene is rendered the object is treated as a transparent polygon and texture mapped. Note that texture mapping takes into account the current view transformation and thus the impostor is warped slightly from frame to frame. Those pixels covered by the transparent pixels are unaffected in value or *z* depth. For the opaque pixels the impostor is treated as a normal polygon and the Z-buffer updated with its depth.

In Maciel and Shirley (1995) 'view-dependent impostors' are pre-calculated – one for each face of the object's bounding box. Space around the object is then divided into view point regions by frustums formed by the bounding box faces and its centre. If an impostor is elected as an appropriate representation then whatever region the current view point is in determines the impostor used.

16.1.2 Calculating the validity of planar impostors

As we have implied, the use of impostors requires an error metric to be calculated to quantify the validity of the impostor. Impostors become invalid because we do not use depth information. At some view point away from the view point from which the impostor was generated the impostor is perceived for what it is – a flat image embedded in three-dimensional space – the illusion is destroyed.

The magnitude of the error depends on the depth variation in the region of the scene represented by the impostor, the distance of the region from the view point and the movement of the view point away from the reference position from which the impostor was rendered. (The distance factor can be gainfully exploited by using lower resolution impostors for distant objects and grouping more than one object into clusters.) For changing view point applications the validity has to be dynamically evaluated and new impostors generated as required.

Shade *et al.* (1996) use a simple metric based on angular discrepancy. Figure 16.2 shows a two-dimensional view of an object bounding box with the plane of the impostor shown in bold. $\boldsymbol{v}_0$ is the view point for the impostor rendering and $\boldsymbol{v}_1$ is the current view point. $\boldsymbol{x}$ is a point or object vertex which coincides with $\boldsymbol{x}'$ in the impostor view. Whenever the view point changes from $\boldsymbol{v}_0$, $\boldsymbol{x}$ and $\boldsymbol{x}'$ subtend an angle θ and Shade *et al.* calculate an error metric which is the maximum angle over all points $\boldsymbol{x}$.

Schaufler and Sturzlinger's (1996) error metric is based on angular discrepancy related to pixel size and the consideration of two worst cases. First, consider the angular discrepancy due to translation of the view point parallel to the impostor plane (Figure 16.3(a)). This is at a maximum when the view point moves normal to a diagonal of a cube enclosing the bounding box with the impostor plane coincident with the other diagonal. When the view point moves to $\boldsymbol{v}_1$ the points $\boldsymbol{x}'$, $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ should be seen as separate points. The angular discrepancy due to this component of view point movement is then given by the angle θ_{trans} between the vectors $\boldsymbol{v}_1\boldsymbol{x}_1$ and $\boldsymbol{v}_1\boldsymbol{x}_2$. As long as this is less that the angle subtended by a pixel at the view point this error can be tolerated. For a view point moving towards the object we consider the construction in Figure 16.3(b). Here the worst case is the corner of the front face of the cube. When the view point moves in to $\boldsymbol{v}_1$ the points $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ should be seen as separate and the angular discrepancy is given as θ_{size}. An impostor can then be used as:

use_imposter := $(\theta_{\text{trans}} < \theta_{\text{screen}})$ or $(\theta_{\text{size}} < \theta_{\text{screen}})$

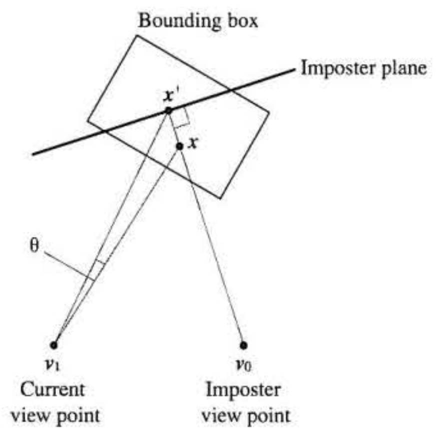
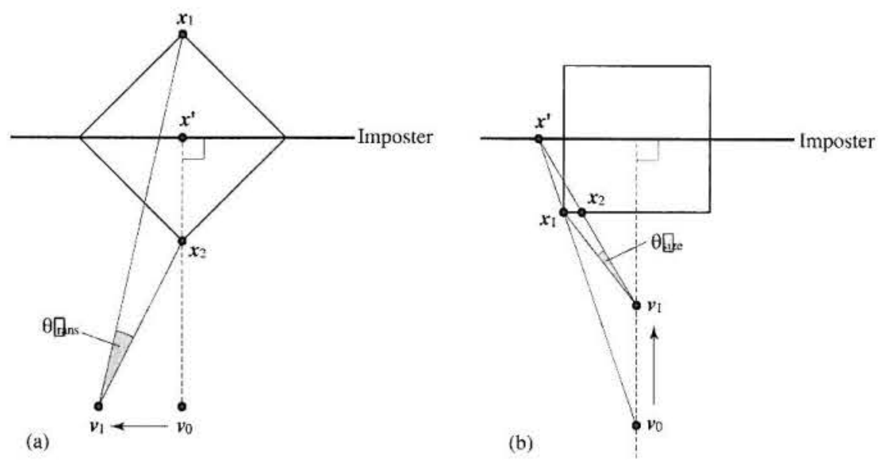


Figure 16.2
Angular discrepancy of an impostor image (after Shade *et al.* (1996)).

Figure 16.3
Schaufler's worst case angular discrepancy metric (after Shaufler (1996)).
(a) Translation of view point parallel to an impostor.
(b) Translation of view point towards an impostor plane.



where:

$$\theta_{\text{screen}} = \frac{\text{field of view}}{\text{screen resolution}}$$

The simplest way to use impostors is to incorporate them as texture maps in a normal rendering scheme exploiting texture mapping hardware.

So far we have said nothing about what makes up an impostor and the assumption has been that we generate an image from an object model. Shade *et al.* (1996) generalize this concept in a scheme called Hierarchical Image Caching and generate impostors from the entire contents of nodes in a BSP tree of the scene combining the benefits of this powerful scene partitioning method with the use of pre-rendered imagery. Thus, for example, distant objects that require infrequent updates can be grouped into clusters and a single impostor generated for the cluster. The algorithm thus operates on and exploits the hierarchy of the scene representation. Objects may be split over different leaf nodes and this leads to the situation of a single objects possessing more than one impostor. This causes visual artefacts and Shade *et al.* (1996) minimize this by ensuring that the BSP partitioning strategy splits as few objects as possible and by 'inflating' the geometry slightly in leaf regions so that the impostors overlap to eliminate gaps in the final image that may otherwise appear.

16.2 Varying rendering resources

16.2.1 Priority rendering

An important technique that has been used in conjunction with 2D imagery is the allocation of different amounts of rendering resources to different parts of the image. An influential (hardware) approach is due to Regan and Pose (1994).

They allocated different frame rates to objects in the scene as a function of their distance from the view point. This was called priority rendering because it combined the environment map approach with updating the scene at different rates. They use a six-view cubic environment map as the basic pre-computed solution. In addition, a multiple display memory is used for image composition and on the fly alterations to the scene are combined with pre-rendered imagery.

The method is a hybrid of a conventional graphics pipeline approach with an image-based approach. It depends on dividing the scene into a priority hierarchy. Objects are allocated a priority depending on their closeness to the current position of the viewer and their allocation of rendering resources and update time are determined accordingly. The scene is pre-rendered as environment maps and, if the viewer remains stationary, no changes are made to the environment map. As the viewer changes position the new environment map from the new view point is rendered according to the priority scheme.

Regan and Pose (1994) utilize multiple display memories to implement priority rendering where each display memory is updated at a different rate according to the information it contains. If a memory contains part of the scene that is being approached by a user then it has to be updated, whereas a memory that contains information far away from the current user position can remain as it is. Thus overall different parts of the scene are updated at different rates – hence priority rendering. Regan and Pose (1994) use memories operating at 60, 30, 15, 7.5 and 3.75 frames per second. Rendering power is directed to those parts of the scene that need it most. At any instant the objects in a scene would be organized into display memories according to their current distance from the user. Simplistically the occupancy of the memories might be arranged as concentric circles emanating from the current position of the user. Dynamically assigning each object to an appropriate display memory involves a calculation which is carried out with respect to a bounding sphere. In the end this factor must impose an upper bound on scene complexity and Regan and Pose (1994) report a test experiment with a test scene of only 1000 objects. Alternatively objects have to be grouped into a hierarchy and dealt with through a secondary data structure as is done in some speed-up approaches to conventional ray tracing.

16.2.2 Image layering

Lengyel and Snyder (1997) generalized the concept of impostors and variable application of rendering resources calling their technique 'coherent image layers'. Here the idea is again to devote rendering resources to different parts of the image according to need expressed as different spatial and/or temporal sampling rates. The technique also deals with objects moving with respect to each other. This is done by dividing the image into layers. (This is, of course, an old idea; since the 1930s cartoon production has been optimized by dividing the image into layers which are worked on independently and composed into a final film.) Thus fast-moving foreground objects can be allocated more resources than slow-moving background objects.

Another key idea of Lengyel and Snyder's work is that any layer can itself be decomposed into a number of components. The layer approach is taken into the shading itself and different resources given to different components in the shading. A moving object may consist of a diffuse layer plus a highlight layer plus a shadow layer. Each component produces an image stream and a stream of two-dimensional transformations representing its translation and warping in image space. Sprites may be represented at different resolutions to the screen resolution and may be updated at different rates. Thus sprites have different resolution in *both* space and time.

A sprite in the context of this work is now an 'independent' entity rather than being a texture map tied to an object by the normal vertex/texture coordinate association. It is also a pure two-dimensional object – not a two-dimensional part (a texture map) of a three-dimensional object. Thus as a sprite moves the appropriate warping has to be calculated.

In effect the traditional rendering pipeline is split into 'parallel' segments each representing a different part of the image (Figure 16.4). Different quality settings can be applied to each layer which manifests in different frame rates and different resolutions for each layer. The layers are then combined in the compositor with transparency or alpha in depth order.

A sprite is created as a rectangular entity by establishing a sprite rendering transform $\boldsymbol{A}$ such that the projection of the object in the sprite domain fits tightly in a bounding box. This is so that points within the sprite do not sample non-object space. The transform $\boldsymbol{A}$ is an affine transform that maps the sprite onto the screen and is determined as follows. If we consider a point in screen space p_s then we have:

$$p_s = \boldsymbol{T}p$$

where p is the equivalent object point in world space and $\boldsymbol{T}$ is the concatenation of the modelling, viewing and projection transformations.

We require an $\boldsymbol{A}$ such that (Figure 16.5):

$$p_s = \boldsymbol{A}^{-1}\boldsymbol{A}\boldsymbol{T}p = \boldsymbol{A}q$$

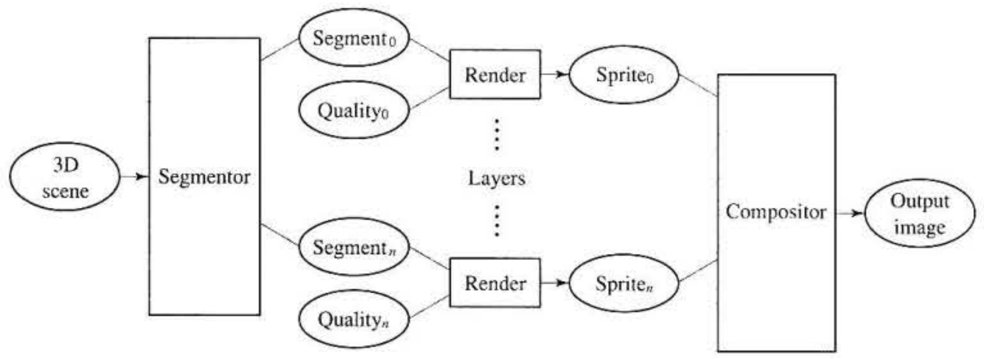


Figure 16.4
The layer approach of Lengyel and Snyder (after Lengyel and Snyder (1997)). Rendering resources are allocated to perceptually important parts of the scene (layers). Slowly changing layers are updated at a lower frame rate and at lower resolution.

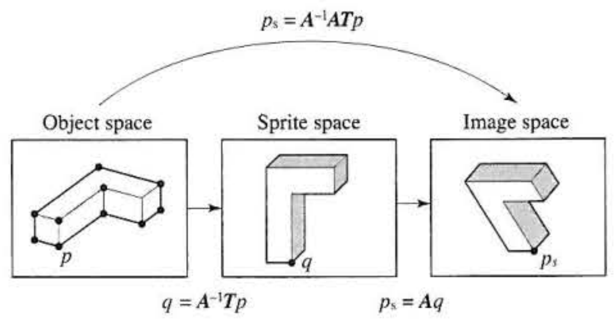


Figure 16.5
The sprite rendering transform ***A***.

where q is a point in sprite coordinates and:

$$\boldsymbol{A} = \begin{bmatrix} a & b & t_x \\ c & d & t_y \end{bmatrix}$$

Thus, an affine transformation is used to achieve an equivalent warp that would occur due to a conventional transformation $\boldsymbol{T}$.

The transform $\boldsymbol{A}$ – a 2 × 3 matrix – is updated as an object undergoes rigid motion and provides the warp necessary to change the shape of the sprite in screen space due to the object motion. This is achieved by transforming the points of a characteristic polyhedron (Figure 16.6) representing the object into screen space for two consecutive time intervals using $\boldsymbol{T}_{n-1}$ and $\boldsymbol{T}_n$ and finding the six unknown coefficients for $\boldsymbol{A}$. Full details of this procedure are given in Lengyel and Snyder (1997).

Calculating the validity of layers

As any sequence proceeds, the reusability of the layers needs to be monitored. In Section 16.1.2 we described a simple geometric measure for the validity of sprites. With image layers Lengyel and Snyder (1997) develop more elaborate criteria based upon geometric, photometric and sampling considerations. The geometric and photometric tests measure the difference between the image due to the layer or sprite and what the image should be if it were conventionally rendered.

A geometric error metric (Lengyel and Snyder call the metrics fiducials) is calculated from:

$$F_{\text{Geometric}} = \max_i \|P_i - \boldsymbol{A}p'_i\|$$

where $\boldsymbol{A}p'_i$ is a set of characteristic points in the layer in the current frame warped into their position from the previous frame and p_i the position the points actually occupy. (These are always transformed by $\boldsymbol{T}$, the modelling, viewing and perspective transform in order to calculate the warp. This sounds like a circular argument but finding $\boldsymbol{A}$ (previous section) involves a best fit procedure. Remember that the warp is being used to approximate the transformation $\boldsymbol{T}$.) Thus a threshold can be set and the layer considered for re-rendering if this is exceeded.

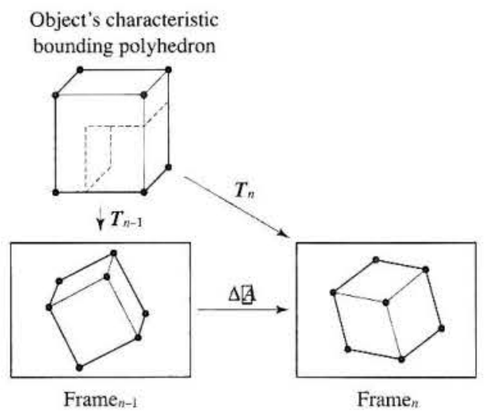


Figure 16.6
The effect of the rigid motion of the points in the bounding polyhedron in screen space is expressed as a change in the affine transform ***A*** (after Lengyel and Snyder (1997)).

For changes due to relative motion between the light source and the object represented by the layer, the angular change in $\boldsymbol{L}$, the light direction vector from the object, can be computed.

Finally, a metric associated with the magnification/minification of the layer has to be computed. If the relative movement between a viewer and object is such that layer samples are stretched or compressed then the layer may need to be re-rendered. This operation is similar to determining the depth parameter in mip-mapping and in this case can be computed from the 2 × 2 sub-matrix of the affine transform.

After a frame is complete a regulator considers resource allocation for the next frame. This can be done either on a 'budget-filling' basis where the scene quality is maximized or on a threshold basis where the error thresholds are set to the highest level the user can tolerate (freeing rendering resources for other tasks). The allocation is made by evaluating the error criteria and estimating the rendering cost per layer based on the fraction of the rendering budget consumed by a particular layer. Layers can then be sorted in a benefit/cost order and re-rendered or warped by the regulator.

Ordering layers in depth

So far nothing has been said about the depth of layers – the compositor requires depth information to be able to generate a final image from the separate layers. Because the method is designed to handle moving objects the depth order of layers can change and the approach is to maintain a sorted list of layers which is dynamically updated. The renderer produces hidden surface eliminated images within a layer and a special algorithm deals with the relative visibility of the layers as indivisible entities. A Kd tree is used in conjunction with convex polyhedra that bound the geometry of the layer and an incremental algorithm (fully described in Snyder (1998)) is employed to deal with occlusion without splitting.

16.3 Using depth information

16.3.1 Three-dimensional warping

As we have already mentioned, the main disadvantage of planar sprites is that they cannot produce motion parallax and they produce a warp that is constrained by a threshold beyond which their planar nature is perceived.

We now come to consider the use of depth information which is, of course, readily available in synthetic imagery. Although the techniques are now going to use the third dimension we still regard them as image-based techniques in the sense that we are still going to use, as source or reference, rendered images albeit augmented with depth information.

Consider, first, what depth information we might employ. The three commonest forms in order of their storage requirements are: using layers or sprites with depth information (previous section), using a complete (unsegmented) image with the associated Z-buffer (in other words one depth value per pixel) and a layered depth image or LDI. An LDI is a single view of a scene with multiple pixels along each line of sight. The amount of storage that LDIs require is a function of the depth complexity of the average number of surfaces that project onto a pixel.

We begin by considering images complete with depth information per pixel – the normal state of affairs for conventionally synthesized imagery. It is intuitively obvious that we should be able to generate or extrapolate an image at a new view point from the reference image providing that the new view point is close to the reference view point. We can define the pixel motion in image space as the warp:

$$I(x, y) \rightarrow I'(x', y')$$

which implies a reference pixel will move to a new destination. (This is a simple statement of the problem which ignores important practical problems that we shall address later.) If we assume that the change in the view point is specified by a rotation $\boldsymbol{R} = [r_{ij}]$ followed by a translation $\boldsymbol{T} = (\Delta x, \Delta y, \Delta z)^{\mathrm{T}}$ of the view coordinate system (in world coordinate space) and that the internal parameters of the viewing system/camera do not change – the focal length is set to unity – then the warp is specified by:

$$x' = \frac{(r_{11}x + r_{12}y + r_{13})Z(x, y) + \Delta x}{(r_{31}x + r_{32}y + r_{33})Z(x, y) + \Delta z}$$

$$y' = \frac{(r_{21}x + r_{22}y + r_{23})Z(x, y) + \Delta y}{(r_{31}x + r_{32}y + r_{33})Z(x, y) + \Delta z} \qquad [16.1]$$

where:

$Z(x, y)$ is the depth of the point $\boldsymbol{P}$ of which (x, y) is the projection.

This follows from:

$$x' = \frac{x_{v'}}{z_{v'}} \qquad y' = \frac{y_{v'}}{z_{v'}}$$

where $(x_{v'}, y_{v'}, z_{v'})$ are the coordinates of the point $\boldsymbol{P}$ in the new viewing system. A visualization of this process is shown in Figure 16.7.

We now consider the problems that occur with this process. The first is called image folding or topological folding and occurs when more than one pixel in the reference image maps into position (x', y') in the extrapolated image (Figure 16.8(a)). The straightforward way to resolve this problem is to calculate $Z(x', y')$ from $Z(x, y)$ but this requires an additional rational expression and an extra Z-buffer to store the results.

McMillan (1995) has developed an algorithm that specifies a unique evaluation order for computing the warp function such that surfaces are drawn in a back-to-front order thus enabling a simple painter's algorithm to resolve this visibility problem. The intuitive justification for this algorithm can be seen by considering a simple special case shown in Figure 16.9. In this case the view point has moved to the left so that its projection in the image plane of the reference view coordinate system is outside and to the left of the reference view window. This fact tells us that the order in which we need to access pixels in the reference is from right to left. This then resolves the problem of the leftmost pixel in the reference image overwriting the right pixel in the warped image. McMillan shows that the accessing or enumeration order of the reference image can be reduced to nine cases depending on the position of the projection of the new view point in the reference coordinate system. These are shown in Figure 16.10. The general case, where the new view point stays within the reference view window divides the image into quadrants. An algorithm structure that utilizes this method to resolve depth problems in the many-to-one case is thus:

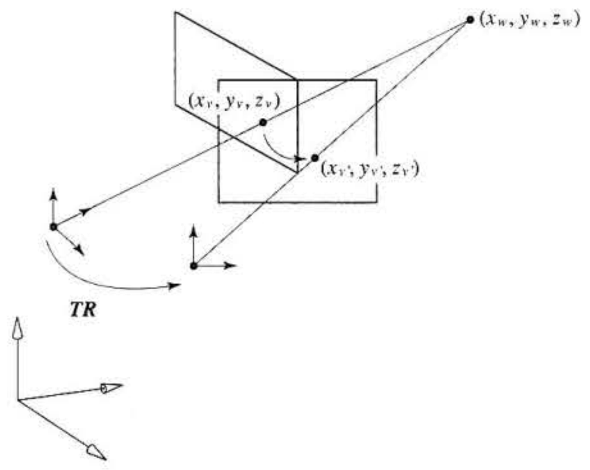


Figure 16.7 A three-dimensional warp is calculated from rotation $\boldsymbol{R}$ and translation $\boldsymbol{T}$ applied to the view coordinate system.

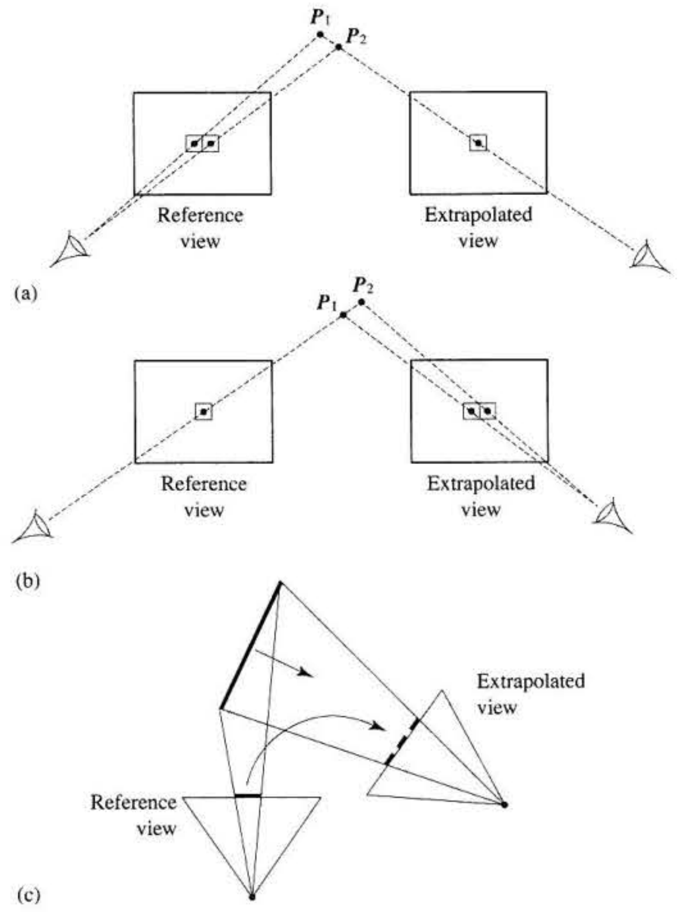


Figure 16.8
Problems in image warping. (a) Image folding: more than one pixel in the reference view maps into a single pixel in the extrapolated view. (b) Holes: information occluded in the reference view is required in the extrapolated view. (c) Holes: the projected area of a surface increases in the extrapolated view because its normal rotates towards the viewing direction. (d) See Colour Plate section.

(1) Calculate the projection of the new view point in the reference coordinate system.

(2) Determine the enumeration order (one out of the nine cases shown in Figure 16.10) depending on the projected point.

(3) Warp the reference image by applying Equation 16.1 and writing the result into the frame buffer.

The second problem produced by image warping is caused when occluded areas in the reference image 'need' to become visible in the extrapolated image (Figure 16.8(b)) producing holes in the extrapolated image. As the figure demonstrates, holes and folds are in a sense the inverse of each other, but where a deterministic solution exists for folds no theoretical solution exists for holes and a heuristic needs to be adopted – we cannot recover information that was not there in the first place. However, it is easy to detect where holes occur. They are simply unassigned pixels in the extrapolated image and this enables the problem to be localized and the most common solution is to fill them in with colours from

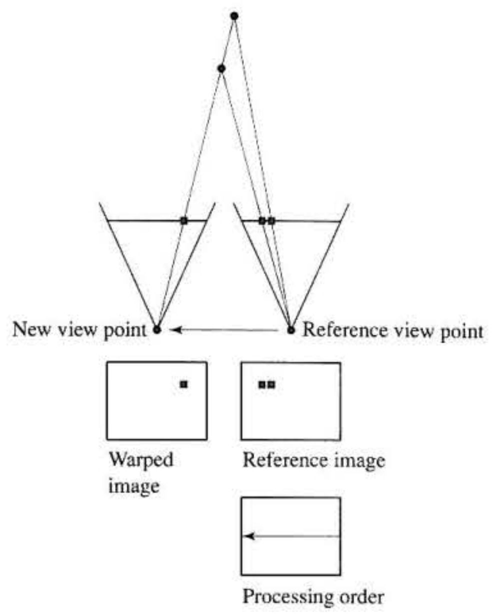


Figure 16.9
The view point translates to the left so that the projection of the new view point in the image plane of the reference view coordinate system is to the left of the reference view window. The correct processing order of the reference pixels is from right to left.

neighbouring pixels. The extent of the holes problem depends on the difference between the reference and extrapolated view points and it can be ameliorated by considering more than one reference image, calculating an extrapolated image from each and compositing the result. Clearly if a sufficient number of reference images are used then the hole problem will be eliminated and there is no need for a local solution which may insert erroneous information.

A more subtle reason for unassigned pixels in the extrapolated image is apparent if we consider surfaces whose normal rotates towards the view direction in

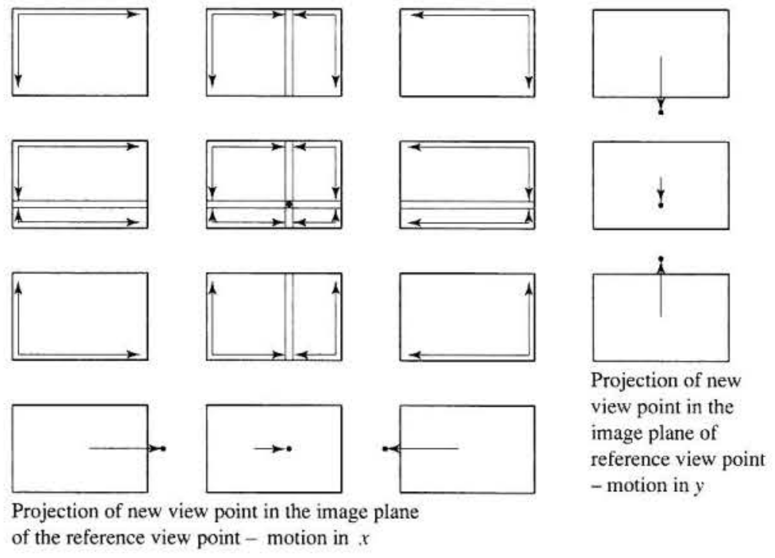

Projection of new view point in the image plane of reference view point – motion in y
Projection of new view point in the image plane of the reference view point – motion in x

Figure 16.10
A visualization of McMillan's priority algorithm indicating the correct processing order as a function of view point motion for nine cases (after McMillan (1995)).

the new view system (Figure 16.8(c)). The projected area of such a surface into the extrapolated image plane will be greater than its projection in the reference image plane and for a one-to-one forward mapping holes will be produced. This suggests that we must take a rigorous approach to reconstruction in the interpolated image. Mark *et al.* (1997) suggest calculating the appropriate dimension of a reconstruction kernel, for each reference pixel as a function of the view point motion but they point out that this leads to a cost per pixel that is greater than the warp cost. This metric is commonly known as splat size (Chapter 13) and its calculation is not straightforward for a single reference image with Z depth only for visible pixels. (A method that stores multiple depth values for a pixel is dealt with in the next section.)

The effects of these problems on an image are shown in Figure 16.8(d) (Colour Plate). The first two images show a simple scene and the corresponding Z-buffer image. The next image shows the artefacts due to translation (only). In this case these are holes caused by missing information and image folding. The next image shows artefacts due to rotation (only) – holes caused by increasing the projected area of surfaces. Note how these form coherent patterns. The final image shows artefacts caused by both rotation and translation.

Finally, we note that view-dependent illumination effects will not in general be handled correctly with this simple approach. This, however, is a problem that is more serious in image-based modelling methods (Section 16.6). As we have already noted in image warping we must have reference images whose view point is close to the required view point.

16.3.2 Layered depth images (LDIs)

Many of the problems encountered in the previous section disappear if our source imagery is in the form of an LDI (Shade *et al.* 1998). In particular we can resolve the problem of holes where we require information in the extrapolated image in areas occluded in the source or reference image. An LDI is a three-dimensional data structure that relates to a particular view point and which samples, for each pixel, all the surfaces and their depth values intersected by the ray through that pixel (Figure 16.11). (In practice, we require a number of LDIs to represent a scene and so can consider a scene representation to be four-dimensional – or the same dimensionality as the light field in Section 16.5.) Thus, each pixel is associated with an array of information with a number of elements or layers that is determined by the number of surfaces intersected. Each element contains a colour, surface normal and depth for surface. Clearly this representation requires much more storage than an image plus Z-buffer but this requirement grows only linearly with depth complexity.

In their work Shade *et al.* (1998) suggest two methods for pre-calculating LDIs for synthetic imagery. First, they suggest warping n images rendered from different view points into a single view point. During the warping process if more than one pixel maps into a single LDI pixel then the depth values associated

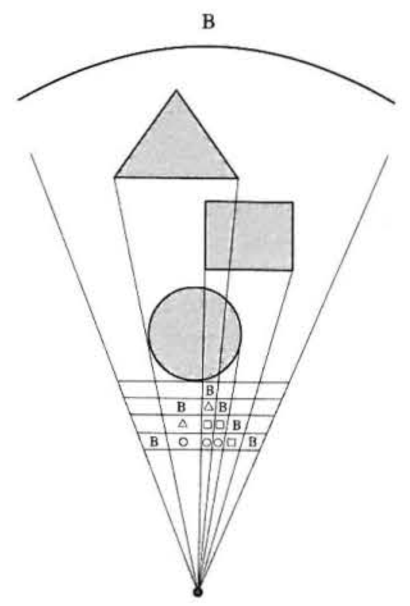


Figure 16.11
A representation of a layered depth image (LDI).

with each source view are compared and enable the layers to be sorted in depth order.

An alternative approach which facilitates a more rigorous sampling of the scene is to use a modified ray tracer. This can be done simplistically by initiating a ray for each pixel from the LDI view point and allowing the rays to penetrate the object (rather than being reflected or refracted). Each hit is then recorded as a new depth pixel in the LDI. All of the scene can be considered by pre-calculating six LDIs each of which consists of a 90° frustum centred on the reference view point. Shade *et al.* (1998) point out that this sampling scheme is not uniform with respect to a hemisphere of directions centred on the view point. Neighbouring pixel rays project a smaller area onto the image plane as a function of the angle between the image plane normal and the ray direction and they weight the ray direction by the cosine of that angle. Thus, each ray has four coordinates: two pixel coordinates and two angles for the ray direction. The algorithm structure to calculate the LDIs is then:

(1) For each pixel, modify the direction and cast the ray into the scene.

(2) For each hit: if the intersected objects lies within the LDI frustum it is re-projected through the LDI view point.

(3) If the new hit is within a tolerance of an existing depth pixel the colour of the new sample is averaged with the existing one; otherwise a new depth pixel is created.

During the rendering phase, an incremental warp is applied to each layer in back to front order and images are alpha blended into the frame buffer without the need for Z sorting. McMillan's algorithm (see Section 16.3.1) is used to ensure

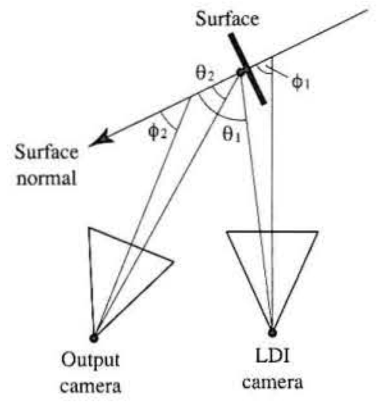


Figure 16.12
Parameters used in splat size computation (after Shade *et al.* (1998)).

that the pixels are selected for warping in the correct order according to the projection of the output camera in the LDI's system.

To enable splat size computation Shade *et al.* (1998) use the following formula (Figure 16.12):

$$\text{size} = \frac{d_1^2 \cos\theta_2 \, \text{res}_2 \tan\dfrac{\text{fov}_1}{2}}{d_2^2 \cos\theta_1 \, \text{res}_1 \tan\dfrac{\text{fov}_2}{2}}$$

where:

size is the dimension of a square kernel (in practice this is rounded to 1, 3, 5 or 7)
the angles θ are approximated as the angles ϕ, where ϕ is the angle between the surface normal and the z axis of the camera system
fov is the field of view of a camera
res = w*h (the width and height of the LDI)

16.4 View interpolation

View interpolation techniques can be regarded as a subset of 3D warping methods. Instead of extrapolating an image from a reference image, they interpolate a pair of reference images. However, to do this three-dimensional calculations are necessary. In the light of our earlier two-dimensional/three-dimensional categorization they could be considered a two-dimensional technique but we have decided to emphasize the interpolation aspect and categorize them separately.

Williams and Chen (1993) were the first to implement view interpolation for a walkthrough application. This was achieved by pre-computing a set of reference images representing an interior – in this case a virtual museum. Frames required in a walkthrough were interpolated at run time from these reference frames. The interpolation was achieved by storing a 'warp script' that specifies the pixel motion between reference frames. This is a dense set of motion vectors that relates a pixel in the source image to a pixel in the destination image. The simplest example of a motion field is that due to a camera translating parallel to its image plane. In that case the motion field is a set of parallel vectors – one for each pixel – with a direction opposite to the camera motion and having a magnitude proportional to the depth of the pixel. This pixel-by-pixel correspondence can be determined for each pair of images since the three-dimensional (image space) coordinates of each pixel is known, as is the camera or view point motion. The determination of warp scripts is a pre-processing step and an interior is finally represented by a set of reference images together with a warp script relating every adjacent pair. For a large scene that requires a number of varied walkthroughs the total storage requirement may be very large; however, any derived or interpolated view only requires the appropriate pair of reference images and the warp script.

At run time a view or set of views between two reference images is then reduced to linear interpolation. Each pixel in both the source and destination images is moved along its motion vector by the amount given by linearly interpolating the image coordinates (Figure 16.13). This gives a pair of interpolated images. These can be composited and using a pair of images in this way reduces the hole problem. Chen and Williams (1993) fill in remaining holes with a procedure that uses the colour local to the hole. Overlaps are resolved by using a Z-buffer to determine the nearest surface, the z values being linearly interpolated along with the (x, y) coordinates. Finally, note that linear interpolation of the motion vectors produces a warp which will not be exactly the same as that produced if the camera was moved into the desired position. The method is only exact from the special case of a camera translating parallel to its image plane. Williams and Chen (1993) point out that a better approximation can be obtained by quadratic or cubic interpolation in the image plane.

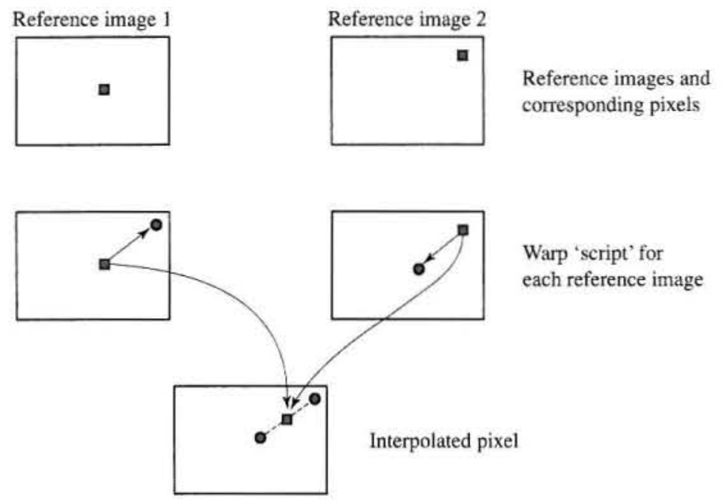

Reference image 1
Reference image 2
Reference images and corresponding pixels
Warp 'script' for each reference image
Interpolated pixel

Figure 16.13
Simple view interpretation: a single pair of corresponding pixels define a path in image space from which an interpolated view can be constructed.

16.4.1 View morphing

Up to now we have considered techniques that deal with a moving view point and static scenes. In a development that they call view morphing Seitz and Dyer (1996) address the problem of generating in-between images where non-rigid transformations have occurred. They do this by addressing the approximation implicit in the previous section and distinguish between 'valid' and 'non-valid' in-between views.

View interpolation by warping a reference image into an extrapolated image proceeds in two-dimensional image plane space. A warping operation is just that – it changes the shape of the two-dimensional projection of objects. Clearly the interpolation should proceed so that the projected shape of the objects in the reference projection is consistent with their real three-dimensional shape. In other words, the interpolated view must be equivalent to a view that would be generated in the normal way (either using a camera or a conventional graphics pipeline) by changing the view point from the reference view point to that of the interpolated view. A 'non-valid' view means that the interpolated view does not preserve the object shape. If this condition does not hold then the interpolated views will correspond to an object whose shape is distorting in real three-dimensional space. This is exactly what happens in conventional image morphing between two shapes. 'Impossible', non-existent or arbitrary shapes occur as in-between images because the motivation here is to appear to change one object into an entirely different one. The distinction between valid and invalid view interpolation is shown in Figure 16.14.

An example where linear interpolation of images produces valid interpolated views is the case where the image planes remain parallel (Figure 16.15). Physically, this situation would occur if a camera was allowed to move parallel to its image plane (and optionally zoom in and out). If we let the combined viewing and perspective transformations (see Chapter 5) be $\boldsymbol{V}_0$ and $\boldsymbol{V}_1$ for the two reference images then the transformation for an in-between image can be obtained by linear interpolation:

$$\boldsymbol{V}_i = (1 - s)\ \boldsymbol{V}_0 + s\boldsymbol{V}_1$$

If we consider a pair of corresponding points in the reference images $\boldsymbol{P}_0$ and $\boldsymbol{P}_1$ which are projections of world space point $\boldsymbol{P}$, then it is easily shown (see Seitz and Dyer (1996)) that the projection of point $\boldsymbol{P}$ from the intermediate (interpolated) view point is given by linear interpolation:

$$\begin{aligned}\boldsymbol{P}_i &= \boldsymbol{P}_0(1 - s) + \boldsymbol{P}_1 s\\ &= \boldsymbol{V}_i\boldsymbol{P}\end{aligned}$$

In other words linear interpolation of pixels along a path determined by pixel correspondence in two reference images is exactly equivalent to projecting the scene point that resulted in these pixels through a viewing and projective transformation given by an intermediate camera position, provided parallel views are maintained, in other words using the transformation $\boldsymbol{V}_i$, which would be obtained if $\boldsymbol{V}_0$ and $\boldsymbol{V}_1$ were linearly interpolated. Note also that we are interpolating views that would correspond to those obtained if we had moved the camera in a straight line from $\boldsymbol{C}_0$ to $\boldsymbol{C}_1$. In other words the interpolated view corresponds to the camera position:

$$\boldsymbol{C}_i = (sC_x, sC_y, 0)$$

If we have reference views that are not related in this way then the interpolation has to be preceded (and followed) by an extra transformation. This is the general situation where the image planes of the reference views and the image plane of the required or interpolated view have no parallel relationship. The first transformation, which Seitz and Dyer call a 'prewarp', warps the reference images so that they appear to have been taken by a camera moving in a plane parallel to its image plane. The pixel interpolation, or morphing, can the be performed as in the previous paragraph and the result of this is postwarped to form the final interpolated view, which is the view required from the virtual camera position.

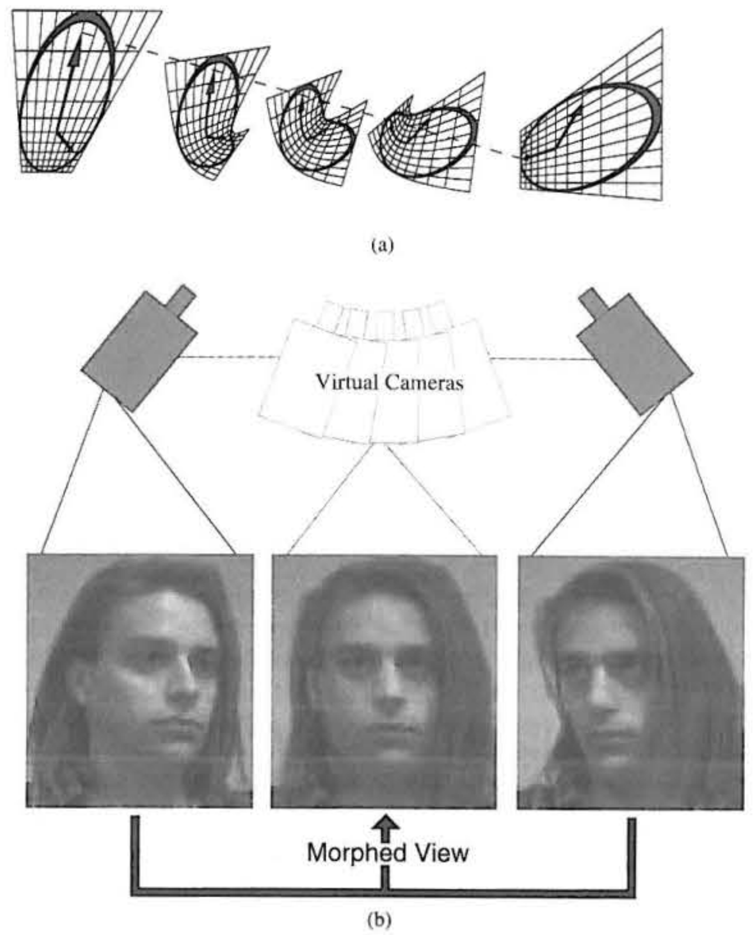


Figure 16.14 Distinguishing between valid and invalid view interpolation. In (a), using a standard (morphing) approach of linear interpolation produces gross shape deformation (this does not matter if we are morphing between two different objects – it becomes part of the effect). (b) The interpolated (or morphed view) is consistent with object shape. (Courtesy of Steven Seitz.)

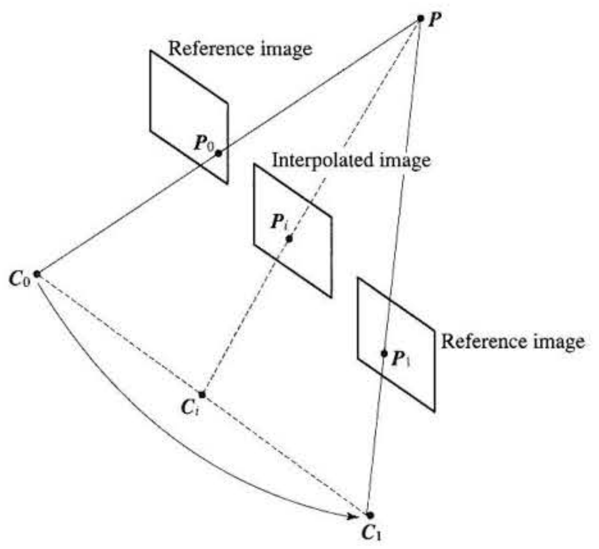
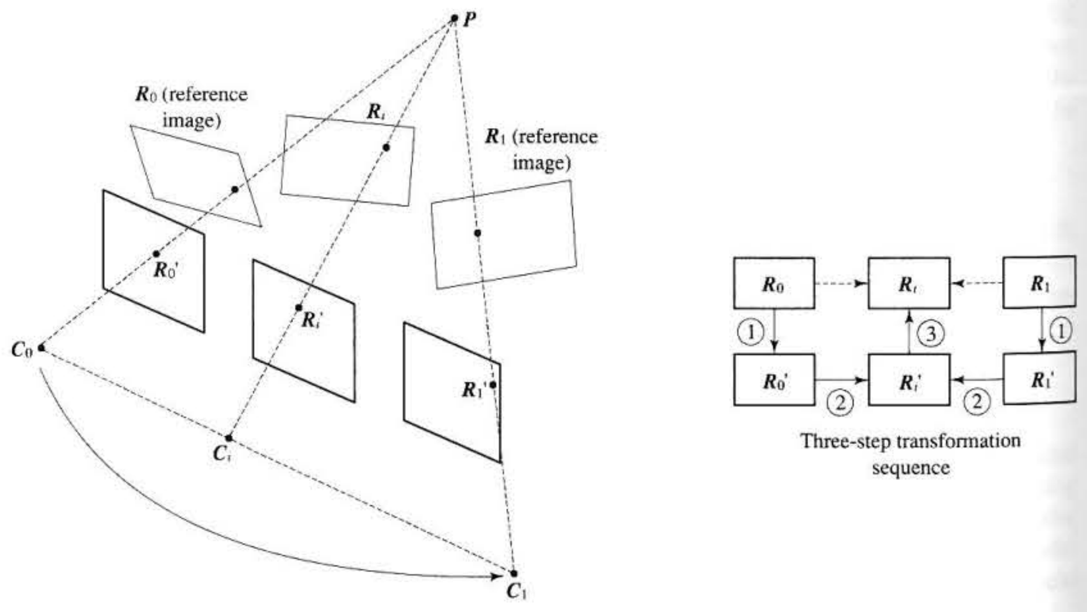


Figure 16.15
Moving the camera from $\boldsymbol{C}_0$ to $\boldsymbol{C}_1$ (and zooming) means that the image planes remain parallel and $\boldsymbol{P}_i$ can be linearly interpolated from $\boldsymbol{P}_0$ and $\boldsymbol{P}_1$ (after Seitz and Dyer (1996)).

A simple geometric illustration of the process is shown in Figure 16.16. Here $\boldsymbol{R}_0$ and $\boldsymbol{R}_1$ are the references images. Prewarping these to $\boldsymbol{R}_0'$ and $\boldsymbol{R}_1'$ respectively means that we can now linearly interpolate these rectified images to produce $\boldsymbol{R}_i'$. This is then postwarped to produce the required $\boldsymbol{R}_i$. An important consequence of this method is that although the warp operation is image based we require knowledge of the view points involved to effect the pre- and post-warp transformations. Again this has ramifications for the context in which the method is used, implying that in the case of photographic imagery we have to record or recover the camera view points.

Figure 16.16
Prewarping reference images, interpolating and postwarping in view interpolation (after Seitz and Dyer (1996)).



The prewarping and postwarping transformations are derived as follows. First, it can be shown that any two perspective views that share the same centre of projection are related by a planar projective transformation – a 3 × 3 matrix obtained from the combined viewing perspective transformation $\boldsymbol{V}$. Thus $\boldsymbol{R}_0$ and $\boldsymbol{R}_1$ are related to $\boldsymbol{R}_0'$ and $\boldsymbol{R}_1'$ by two such matrices $\boldsymbol{T}_0$ and $\boldsymbol{T}_1$. The procedure is thus as follows:

(1) Prewarp $\boldsymbol{R}_0$ and $\boldsymbol{R}_1$ using $\boldsymbol{T}_0^{-1}$ and $\boldsymbol{T}_1^{-1}$ to produce $\boldsymbol{R}_0'$ and $\boldsymbol{R}_1'$.

(2) Interpolate to calculate $\boldsymbol{R}_i'$, $\boldsymbol{C}_i$ and $\boldsymbol{T}_i$.

(3) Apply $\boldsymbol{T}_i$ to $\boldsymbol{R}_i'$ to give image $\boldsymbol{R}_i$.

16.5 Four-dimensional techniques – the Lumigraph or light field rendering approach

Up to now we have considered systems that have used a single image or a small number of reference images from which a required image is generated. We have looked at two-dimensional techniques and methods where depth information has been used – three-dimensional warping. Some of these methods involve pre-calculation of a special form of rendered imagery (LDIs) others post-process a conventionally rendered image. We now come to a method that is an almost total pre-calculation technique. It is an approach that bears some relationship to environment mapping. An environment map caches all the light rays that arrive at a single point in the scene – the source or reference point for the environment map. By placing an object at that point we can (approximately) determine those light rays that arrive at the surface of the object by indexing into the map. This scheme can be extended so that we store in effect an environment map for every sampled point in the scene. That is, for each point in the scene we have knowledge of all light rays arriving at that point. We can now place an object at any point in the scene and calculate the reflected light. The advantage of this approach is that we now minimize most of the problems related to three-dimensional warping at the cost of storing a vast amount of data.

A light field is a similar approach. For each and every point of a region in the scene in which we wish to reconstruct a view we pre-calculate and store or cache the radiance in every direction at that point. This representation is called a light field or Lumigraph (Levoy and Hanrahan 1996; Gortler *et al.* 1996) and we construct for a region of free space by which is meant a region free of occluders. The importance of free space is that it reduces the light field from a five-dimensional to a four-dimensional function. In general, for every point (x, y, z) in scene space we have light rays travelling in every direction (parametrized by two angles) giving a five-dimensional function. In occluder free space we can assume (unless there is atmospheric interaction) that the radiance along a ray is constant. The two 'free space scenes' of interest to us are: viewing an object from anywhere outside its convex hull and viewing an environment such as a room from somewhere within its (empty) interior.

The set of rays in any region in space can be parametrized by their intersection with two parallel planes and this is the most convenient representation for a light field (Figure 16.17(a)). The planes can be positioned anywhere. For example, we can position a pair of planes parallel to each face of a cube enclosing an object and capture all the radiance information due to the object (Figure 16.7(b)). Reconstruction of any view of the object then consists of each pixel in the view plane casting a ray through the plane pair and assigning $L(s, t, u, v)$ to that pixel (Figure 16.7(c)). The reconstruction is essentially a resampling process and unlike the methods described in previous sections it is a linear operation.

Light fields are easily constructed from rendered imagery. A light field for a single pair of parallel planes placed near an object can be created by moving the camera in equal increments in the (s, t) plane to generate a series of sheared perspective projections. Each camera point (s, t) then specifies a bundle of rays arriving from every direction in the frustum bounded by the (u, v) extent. It could be argued that we are simply pre-calculating every view of the object that we require at run time; however, two factors mitigate this brute-force approach. First, the resolution in the (s, t) plane can be substantially lower than the resolution in the (u, v) plane. If we consider a point on the surface of the object coincidence, say, with the (u, v) plane, then the (s, t) plane contains the reflected

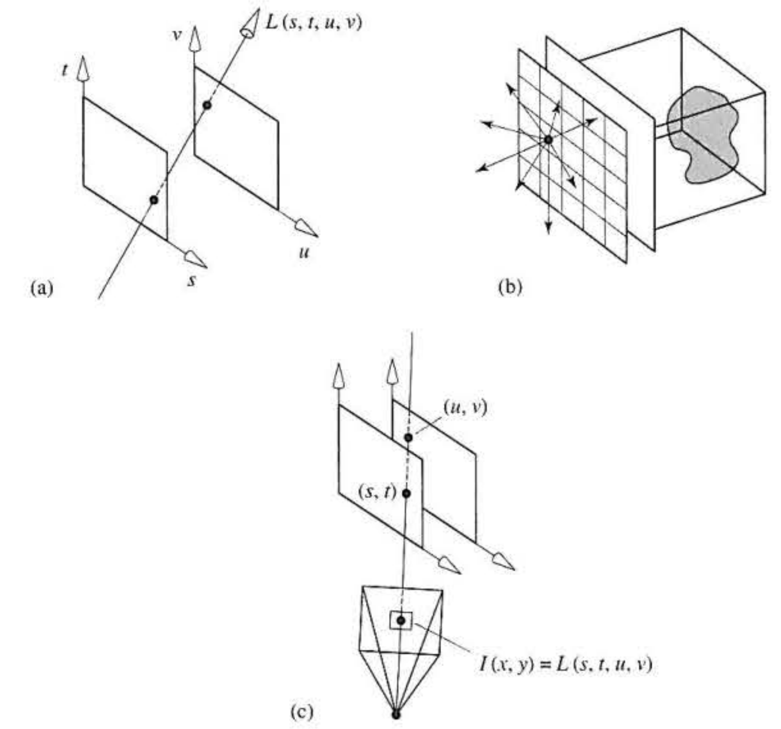


Figure 16.17
Light field rendering using parallel plane representation for rays. (a) Parametrization of a ray using parallel planes. (b) Pairs of planes positioned on the face of a bounding cube can represent all the radiance information due to an object. (c) Reconstruction for a single pixel $I(x,y)$.

light in every direction (constrained by the (s, t) plane extent). By definition, the radiance at a single point on the surface of an object varies slowly with direction and a low sampling frequency in the (s, t) plane will capture this variation. A higher sampling frequency is required to calculate the variation as a function of position on the surface of the object. Second, there is substantial coherence exhibited by a light field. Levoy and Hanrahan (1996) report a compression ratio of 118:1 for a 402 Mb light field and conclude that given this magnitude of compression the simple (linear) re-sampling scheme together with simplicity advantages over other IBR methods make light fields a viable proposition.

16.6 Photo-modelling and IBR

Another distinguishing factor in IBR approaches is whether they work only with computer graphics imagery (where depth information is available) or whether they use photographs as the source imagery. Photography has the potential to solve the other major problem with scene complexity – the modelling cost. Real world detail, whose richness and complexity eludes even the most elaborate photo-realistic renderers, is easily captured by conventional photographic means. The idea is to use IBR techniques to manipulate the photographs so that they can be used to generate an image from a view point different from the camera view point.

Photographs have always been used in texture mapping and this classical tool is still finding new applications in areas which demand an impression of realism that would be unobtainable from conventional modelling techniques, except at great expense. A good example is facial animation where a photograph of a face is wrapped onto a computer graphics model or structure. The photo-map provides the fine level of detail, necessary for convincing and realistic expressions, and the underlying three-dimensional model is used as a basis for controlling the animation.

In building geometric representations from photographs, many of the problems that are encountered are traditionally part of the computer vision area but the goals are different. Geometric information recovered from a scene in a computer vision context usually has some single goal, such as collision avoidance in robot navigation or object recognition, and we are usually concerned in some way with reducing the information that impinges on the low-level sensor. We are generally interested in recovering the shape of an object without regard to such irrelevant information as texture; although we may use such information as a device for extracting the required geometry, we are not interested in it *per se*. In modelling a scene in detail, it is precisely the details such as texture that we are interested in, as well as the pure geometry.

Consider first the device of using photography to assist in modelling. Currently available commercial photo-modelling software concentrates on extracting pure geometry using a high degree of manual intervention. Common approaches use a pre-calibrated camera, knowledge of the position of the camera for each shot and a sufficient number of shots to capture the structure of the building, say, that is being modelled. Extracting the edges from the shots of the building enables a wireframe model to be constructed. This is usually done semi-automatically with an operator matching corresponding edges in the different projections. It is exactly equivalent to the shape from stereo problem using feature correspondence except that now we use a human being instead of a correspondence-establishing algorithm. We may end up performing a large amount of manual work on the projections, as much work as would be entailed in using a conventional CAD package to construct the building. The obvious potential advantage is that photo-modelling offers the possibility of automatically extracting the rich visual detail of the scene, as well as the geometry.

It is interesting to note that in modelling from photographs approaches, the computer graphics community has side-stepped the most difficult problems that are researched in computer vision by embracing some degree of manual intervention. For example, the classical problem of correspondence between images projected from different view points is solved by having an operator manually establish a degree of correspondence between frames which can enable the success of algorithms that establish detailed pixel-by-pixel correspondence. In computer vision such approaches do not seem to be considered. Perhaps this is due to well-established traditional attitudes in computer vision which has tended to see the imitation of human capabilities as an ultimate goal, as well as constraints from applications.

Using photo-modelling to capture detail has some problems. One is that the information we obtain may contain light source and view-dependent phenomena such as shadows and specular reflections. These would have to be removed before the imagery could be used generate the simulated environment from any view point. Another problem of significance is that we may need to warp detail in a photograph to fit the geometric model. This may involve expanding a very small area of an image. Consider, for example, a photograph – taken from the ground – of high building with a detailed facade. Important detail information near the top of the building may be mapped into a small area due to the projective distortion. In fact, this problem is identical to view interpolation.

Let us now consider the use of photo-modelling without attempting to extract the geometry. We simply keep the collected images as two-dimensional projections and use these to calculate new two-dimensional projections. We never attempt to recover three-dimensional geometry of the scene (although it is necessary to consider the three-dimensional information concerning the projections). This is a form of image-based rendering and it has something of a history.

Consider a virtual walk through an art gallery or museum. The quality requirements are obvious. The user needs to experience the subtle lighting conditions designed to best view the exhibits. These must be reproduced and sufficient detail must be visible in the paintings. A standard computer graphics approach may result in using a (view-independent) radiosity solution for the rendering together with (photographic) texture maps for the paintings. The

radiosity approach, where the expensive rendering calculations are performed once only to give a view-independent solution may suffice in many contexts in virtual reality, but it is not a general solution for scenes that contain complex geometrical detail. As we know, a radiosity rendered scene has to be divided up into as large elements as possible to facilitate a solution and there is always a high cost for detailed scene geometry.

This kind of application – virtual tours around buildings and the like – has already emerged with the bulk storage freedom offered by videodisk and CD-ROM. The inherent disadvantage of most approaches is that they do not offer continuous movement or walkthrough but discrete views selected by a user's position as he (interactively) navigates around the building. They are akin to an interactive catalogue and require the user to navigate in discrete steps from one position to the other as determined by the points from which the photographic images were taken. The user 'hops' from view point to view point.

An early example of a videodisk implementation is the 'Movie Map' developed in 1980 (Lippman 1980). In this early example the streets of Aspen were filmed at 10-foot intervals. To invoke a walkthrough, a viewer retrieved selected views from two videodisk players. To record the environment four cameras were used at every view point – thus enabling the viewer to pan to the left and right. The example demonstrates the trade-off implicit in this approach – because all reconstructed views are pre-stored the recording is limited to discrete view points.

An obvious computer graphics approach is to use environment maps – originally developed in rendering to enable a surrounding environment to be reflected in a shiny object (see Chapter 8). In image-based rendering we simply replace the shiny object with a virtual viewer. Consider a user positioned at a point from which a six-view (cubic) environment map has been constructed (either photographically or synthetically). If we make the approximation that the user's eyes are always positioned exactly at the environment map's view point then we can compose any view direction-dependent projection demanded by the user changing his direction of gaze by sampling the appropriate environment maps. This idea is shown schematically in Figure 16.18. Thus we have, for a stationary viewer, coincidentally positioned at the environment map view point, achieved our goal of a view-independent solution. We have decoupled the viewing direction from the rendering pipeline. Composing a new view now consists of sampling environment maps and the scene complexity problem has been bound by the resolution of the pre-computed or photographed maps.

The highest demand on an image generator used in immersive virtual reality comes from head movements (we need to compute at 60 frames per second to avoid the head latency effect) and if we can devise a method where the rendering cost is almost independent of head movement this would be a great step forward. However, the environment map suggestion only works for a stationary viewer. We would need a set of maps for each position that the viewer could be in. Can we extend the environment map approach to cope with complete walkthroughs? Using the constraint that in a walkthrough the eyes of the user are always at a constant height, we could construct a number of environment maps

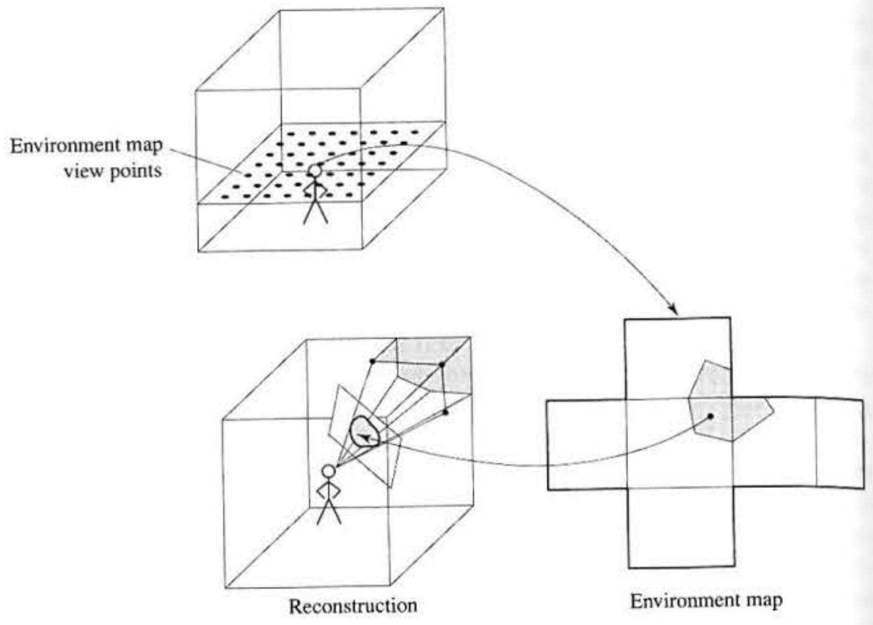


Figure 16.18
Compositing a user projection from an environment map.

whose view points were situated at the lattice points of a coarse grid in a plane, parallel to the ground plane and positioned at eye height. For any user position could we compose, to some degree of accuracy, a user projection by using information from the environment maps at the four adjacent lattice points? The quality of the final projections are going to depend on the resolution of the maps and the number of maps taken in a room – the resolution of the eye plane lattice. The map resolution will determine the detailed quality of the projection and the number of maps its geometric accuracy.

To be able to emulate the flexibility of using a traditional graphics pipeline approach, by using photographs (or pre-rendered environment maps), we either have to use a brute-force approach and collect sufficient views compatible with the required 'resolution' of our walkthrough, or we have to try to obtain new views from the existing ones.

Currently, viewing from cylindrical panoramas is being established as a popular facility on PC-based equipment (see Section 16.5.1). This involves collecting the component images by moving a camera in a semi-constrained manner – rotating it in a horizontal plane. The computer is used merely to 'stitch' the component images into a continuous panorama – no attempt is made to recover depth information.

This system can be seen as the beginning of development that may eventually result in being able to capture all the information in a scene by walking around with a video camera resulting in a three-dimensional photograph of the scene. We could see such a development as merging the separate stages of modelling and rendering, there is now no distinction between them. The virtual viewer can then be immersed in a photographic-quality environment and have the freedom to move around in it without having his movement restricted to the excursions of the camera.

16.6.1 Image-based rendering using photographic panoramas

Developed in 1994, Apple's QuickTime® VR is a classic example of using a photographic panorama as a pre-stored virtual environment. A cylindrical panorama is chosen for this system because it does not require any special equipment beyond a standard camera and a tripod with some accessories. As for re-projection – a cylindrical map has the advantage that it only curves in one direction thus making the necessary warping to produce the desired planar projection fast. The basic disadvantage of the cylindrical map – the restricted vertical field of view – can be overcome by using an alternative cubic or spherical map but both of these involve a more difficult photographic collection process and the sphere is more difficult to warp. The inherent viewing disadvantage of the cylinder depends on the application. For example, in architectural visualization it may be a serious drawback.

Figure 16.19 (Colour Plate) is an illustration of the system. A user takes a series of normal photographs, using a camera rotating on a tripod, which are then 'stitched' together to form a cylindrical panoramic image. A viewer positions himself at the view point and looks at a portion of the cylindrical surface. The re-projection of selected part of the cylinder onto a (planar) view surface involves a simple image warping operation which, in conjunction with other speed-up strategies, operates in real time on a standard PC. A viewer can continuously pan in the horizontal direction and the vertical direction to within the vertical field of view limit.

Currently restricted to monocular imagery, it is interesting to note that one of the most lauded aspects of virtual reality – three-dimensionality and immersion – has been for the moment ignored. It may be that in the immediate future monocular non-immersive imagery, which does not require expensive stereo viewing facilities and which concentrates on reproducing a visually complex environment, will predominate in the popularization of virtual reality facilities.

16.6.2 Compositing panoramas

Compositing environment maps with synthetic imagery is straightforward. For example, to construct a cylindrical panorama we map view space coordinates (x, y, z) onto a cylindrical viewing surface (θ, h) as:

$$\theta = \tan^{-1}(x/z) \qquad h = y/(x^2 + z^2)^{1/2}$$

Constructing a cylindrical panorama from photographs involves a number of practical points. Instead of having three-dimensional coordinates we now have photographs. The above equations can still be used substituting the focal length of the lens for z and calculating x and y from the coordinates in the photograph plane and the lens parameters. This is equivalent to considering the scene as a picture of itself – all objects in the scene are considered to be at the same depth.

Another inherent advantage of a cylindrical panorama is that after the overlapping planar photographs are mapped into cylindrical coordinates (just as if we had a cylindrical film plane in the camera) the construction of the complete panorama can be achieved by translation only – implying that it is straightforward to automate the process. The separate images are moved over one another until a match is achieved – a process sometimes called 'stitching'. As well as translating the component images, the photographs may have to be processed to correct for exposure differences that would otherwise leave a visible vertical boundary in the panorama.

The overall process can now be seen as a warping of the scene onto a cylindrical viewing surface followed by the inverse warping to re-obtain a planar projection from the panorama. From the user's point of view the cylinder enables both an easy image collection model and a natural model for viewing in the sense that we normally view an environment from a fixed height – eye level – look around and up and down.

16.6.3 Photo-modelling for image-based rendering

In one of the first comprehensive studies of photo-modelling for image-based rendering, Debevec *et al.* (1996) describe an approach with a number of interesting and potentially important features. Their basic approach is to derive sufficient information from sparse views of a scene to facilitate image-based rendering (although the derived model can also be used in a conventional rendering system). The emphasis of their work is architectural scenes and is based on three innovations:

(1) **Photogrammetric modelling** in which they recover a three-dimensional geometric model of a building based on simple volumetric primitives together with the camera view points from a set of sparse views.

(2) **View-dependent texture mapping** which is used to render the recovered model.

(3) **Model-based stereo** which is used to solve the correspondence problem (and thus enable view interpolation) and the recovery of detail not modelled in (1).

Debevec *et al.* (1996) state that their approach is successful because:

it splits the task of modelling from images into tasks that are easily accomplished by a person (but not by a computer algorithm), and tasks which are easily performed by a computer algorithm (but not by a person).

The photogrammetric modelling process involves the user viewing a set of photographs of a building and associating a set of volumetric primitives with the photographic views to define an approximate geometric model. This is done by invoking a component of the model, such as a rectangular solid and interactively associating edges in the model with edges in the scene. In this way a box, say, can be fitted semi-automatically to a view or views that contain a box as a structural element. This manual intervention enables a complete geometric model to be derived from the photographs even though only parts of the model may be visible in the scene. The accuracy of the geometric model – that is the difference between the model and the reality – depends on how much detail the user invokes, the nature of the volumetric primitives and the nature of the scene. The idea is to obtain a geometric model that reflects the structure of the building and which can be used in subsequent processing to derive camera positions and facilitate a correspondence algorithm. Thus a modern tower block may be represented by a single box, and depth variations, which occur over a face due to windows that are contained in a plane parallel to the wall plane, are at this stage of the process ignored.

Once a complete geometric model has been defined, a reconstruction algorithm is invoked, for each photographic view. The purpose of this process is to recover the camera view points, which is necessary for view interpolation, together with the world coordinates of the model, which are necessary if the model is going to be used in a conventional rendering system. This is done by projecting the geometric model, using a hypothesized view point, onto the photographic views and comparing the position of the image edges with the position of the projected model edges. The algorithm works by minimizing an objective function which operates on the error between observed image edges and the projected model edges. Correspondence between model edges and image edges having already been established, the algorithm has to proceed towards the solution without getting stuck at a local minimum.

These two processes – photogrammetric modelling and reconstruction – extract sufficient information to enable a conventional rendering process that Debevec calls 'view-dependent texture mapping'. Here a new view of a building is generated by projecting the geometric model from the required view point, treating the reference views as texture maps and reprojecting these from the new view point onto the geometric model. The implication here is that the building is 'oversampled' and any one point will appear in two or more photographic views. Thus when a new or virtual view is generated there will, for each pixel in the new view, be a choice of texture maps with (perhaps) different values for the same point on the building due to specularities and unmodelled geometric detail. This problem is approached by mixing the contributions in inverse proportion to the angles that the new view makes with the reference view directions as shown in Figure 16.20. Hence the term 'view-dependent texture mapping' – the contributions are selected and mixed according to the position of the virtual view point with respect to the reference views.

The accuracy of this rendering is limited to the detail captured by the geometric model and there is a difference between the real geometry and that of the

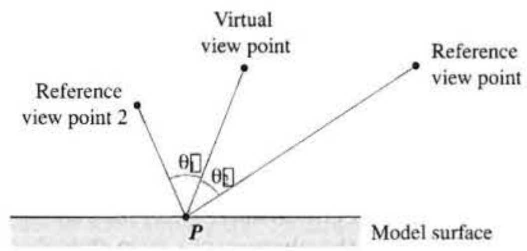


Figure 16.20
The pixel that corresponds to point ***P*** in the virtual view receives a weighted average of the corresponding pixels in the reference images. The weights are inversely proportional to θ_1 and θ_2.

model. The extent of this difference depends on the labour that the user has put into the interactive modelling phase and the assumption is that the geometric model will be missing such detail as window recesses and so on. For example, a facade modelled as a plane may receive a texture that contains such depth information as shading differences and this can lead to images that do not look correct. The extent of this depends on the difference between the required viewing angle and the angle of the view from which the texture map was selected. Debevec *et al.* (1996) go on to extend their method by using the geometric model to facilitate a correspondence algorithm that enables a depth map to be calculated and the geometric detail missing from the original model to be extracted. Establishing correspondence also enables view interpolation.

This process is called 'model-based' stereo and it uses the geometric model as *a priori* information which enables the algorithm to cope with views that have been taken from relatively far apart – one of the practical motivations of the work is that it operates with a sparse set of views. (The main problem with traditional stereo correspondence algorithms is that they try to operate without prior knowledge of scene structure. Here the extent of the correspondence problem predominantly depends on how close the two views are to each other.)

17 Computer animation

Computer animation is a huge subject and deserves a complete textbook in its own right. This chapter concentrates on foundation topics that have become established in the field and serves as an introduction to the subject, rather than comprehensive coverage. The aim of the material is to give the reader a good grounding in the concepts on which most modern systems are based.

Introduction

Leaving aside some toys of the nineteenth century, it is interesting to consider that we have only had the ability to create and disseminate moving imagery for a very short period – since the advent of film. In this time it seems that animation has not developed as a mainstream art form. Outside the world of Disney and his imitators there is little film animation that reaches the eyes of the common man. It is curious that we do not seem to be interested in art that represents movement and mostly consign animation to the world of children's entertainment. Perhaps we can thank Disney for raising film animation to an art form and at the same time condemning it to a strange world of cute animals who are imbued with a set of human emotions.